Надежда Григорьевна Голубь

Искусство программирования на АССЕМБЛЕРЕ

лекции и упражнения вторая редакция

Воспользуйтесь советами программиста-профессионала

Разберитесь в особенностях работы компьютера

Подробно рассмотрены

- Системы счисления
- Внутреннее представление данных в компьютере IBM PC
- Принципы программирования на Ассемблере: команды, директивы, регистры, распределение оперативной памяти, способы адресации
- Соглашения о стыковке разноязычных модулей: Паскаль и Ассемблер, C/C++ и Ассемблер, особенности и хитрости их программирования
- Основные принципы тестирования и отладки программ
- Основные команды целочисленного программирования на Ассемблере
- Команды сопроцессора
- Машинный код
- Особенности 16- и 32-разрядного программирования
- Принципы организации ввода-вывода информации
- Программирование в DOS и Windows



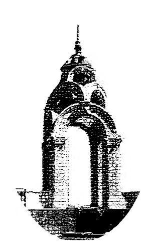
книга-почтой интернет-магазин
www.diasoft.kiev.ua

Голубь Н.Г.

Искусство программирования на Ассемблере

Лекции и упражнения

ИЗДАНИЕ ВТОРОЕ
исправленное и дополненное





Москва • Санкт-Петербург • Киев
2002

ББК 32.973.2
УДК 681.3. 06(075)

Г 26

Г 26 Искусство программирования на Ассемблере. Лекции и упражнения: Голубь Н.Г.— 2-е изд., испр. и доп. — СПб.: ООО «ДиаСофтЮП», 2002.— 656 с.

ISBN 5-93772-056-3

В книге дано описание основных элементов языка Ассемблера семейства IBM PC: системы счисления, машинное представление данных и команд, основы 16- и 32-разрядного программирования, программирование сопроцессора, ввод-вывод информации в DOS и Windows, использование макросредств. Подробно, шаг за шагом на многочисленных примерах законченных программ рассматриваются идеи и принципы организации вычислений на Ассемблере от простого к сложному, используя аналогию и прямую поддержку со стороны алгоритмических языков Pascal (Borland Pascal-7.0, Delphi-5) и C/C++ (Borland C++ 3.1, 4.5, 5.02, Borland C++ Builder 5, Visual C++ 6.0).

Материал книги базируется на лекционном курсе и лабораторном практикуме "Основы организации и функционирования ЭВМ" в течение многих лет читаемого автором для программистов в Национальном аэрокосмическом университете имени Н.Е. Жуковского (ХАИ). Книга состоит из двух частей: лекции и лабораторные работы. Каждая лабораторная работа содержит подробно разобранные варианты решения типовой задачи с указанием возможных проблем при вычислениях и способах их устранения.

Книга содержит необходимый справочный материал, большое количество примеров и законченных программ. Все исходные тексты программ находятся на прилагаемой к книге дискете, а более расширенная информация — на сайте издательства.

Для разработчиков программного обеспечения, желающих повысить качество своих программ, преподавателей и студентов, профессионально изучающих программирование, а также для всех желающих познакомиться с нижним уровнем программирования компьютеров семейства IBM PC.

ББК 32.973.2

Гигиеническое заключение № 77.99.6.953.П.438.2.99 от 04.02.1999

Содержание

ВСЕМ МОИМ УЧИТЕЛЯМ И УЧЕНИКАМ
ПОСВЯЩАЕТСЯ

Введение

Зачем собственно нужен **Ассемблер** в современных условиях столь бурного развития программных средств, призванных существенно облегчить жизнь программиста? Чтобы это понять, рассмотрим вкратце историю появления и развития основных **процедурных** алгоритмических языков и стилей программирования.

Исторический экскурс

В основе того или иного языка программирования лежит некая руководящая идея, вызванная потребностями или, чаще всего, *кризисом* в области программирования и создания программного обеспечения, которая оказывает существенное влияние на стиль программирования и помогает преодолеть указанный кризис.

1. **Машинно-ориентированное программирование** появилось одновременно с созданием электронных вычислительных машин. Сначала это были программы в машинных кодах, затем появился язык программирования **Assembler (Автокод)**, который немного "очеловечил" написание программы в машинном коде. Этот стиль программирования предполагает *доскональное знание возможностей конкретной архитектуры ЭВМ и операционной системы* и используется до сих пор тогда, когда другие стили бессильны, или нужно получить максимальное быстродействие в рамках той или иной операционной системы с использованием архитектуры данной ЭВМ.

2. **Процедурное программирование**. Основная идея этого стиля — ***алгоритмизация** процесса решения задачи и выбор наилучшего алгоритма* (по расходу памяти или по быстродействию). Реализация этой идеи началась с 1957 года с появлением алгоритмических языков **Fortran** и затем **Algol-60**, когда все большее и большее число специалистов (сначала

это были, конечно, математики) занялось решением достаточно сложных инженерных и научных задач. И нужен был стиль программирования максимально близкий к человеческому (математическому) стилю. При этом знание тонкостей архитектуры ЭВМ не требовалось. Программа на алгоритмическом языке (при наличии соответствующих трансляторов) должна была в идеале работать на ЭВМ любой архитектуры. Но это были программы прикладные. Разработку же системных программ (самих трансляторов, систем ввода-вывода) по-прежнему надо было делать на Ассемблере.

3. **Структурное программирование**. Здесь основная идея прекрасно выражена Н. Виртом в его книге "***Алгоритмы + структуры данных = программы***". Это был ответ на кризис в области программирования, начавшийся в середине 60-х годов, когда объем исходного программного кода перешел рубеж в 1000 строк. В 1971 году появился алгоритмический язык **Pascal** и немного позже, в 1972 году, язык **C**. Алгоритмические языки стали более мощными, более "интеллектуальными", на них уже можно было писать элементы трансляторов (компиляторов) и драйверов (подпрограмм обработки ввода-вывода). Компиляторы с языков **C** и **Fortran** выдают, например, по требованию программиста и листинг программы на Ассемблере. *Знающий Ассемблер программист* может его проанализировать, что-то подправить (а, как вы и сами увидите в дальнейшем, подправлять есть что!) и перекомпилировать, но уже на Ассемблере. В этом случае можно достаточно быстро и эффективно получать системные программы.

4. **Модульное программирование**. Здесь основная идея заключалась в том, чтобы "***спрятать***" *данные и процедуры внутри независимых программных единиц — модулей*. Эту идею впервые реализовал Н. Вирт в алгоритмическом языке **Modula** (1975-1979 годы), а затем "подхватили" и остальные, распространенные в то время языки программирования. Например, известные системы программирования **Turbo Pascal** и **Turbo C**.

5. **Объектно-ориентированное программирование**. С середины 80-х годов объем исходного программного кода перешел рубеж в 100 000 строк. Нужно было сделать не случайное объединение данных и алгоритмов их обработки в единое целое, а — смысловое. То есть необходимо было создать *модульное программирование нового уровня*, когда основной акцент делается *на* ***смысловую связь*** *структур данных и алгоритмов их обработки*. Сейчас практически **все основные языки программирования (их более 100,** в том числе такие распространенные, как **Object Pascal, C++, Java, Smalltalk)** базируются на этой идее, а предком их является язык **Simula**, созданный еще в 1960 году.

6. **Обобщенные технологии разработки приложений**. Идеология объектно-ориентированного программирования породила **CASE**-технологии разработки и сборки программ на основе уже *известных* программных моделей, содержащих ***интерфейсы*** и ***прототипы*** (шаблоны — template) ***данных***: **COM** (Component Object Model), **STL** (Standard Template Library), **ATL** (Active Template Library). Все эти новшества поддерживают ***визуальные*** среды разработки, например, такие известные, как **Visual C++, Borland C++ Builder**.

Зачем это нужно

А нужно ли знание **Ассемблера** обычному, прикладному программисту? Если он хочет *понимать*, почему его программа, написанная на алгоритмических (высокоуровневых) языках в любом из стилей 2-6, иногда нестабильно и даже непредсказуемо (с точки зрения пользователя) работает, конечно, нужно. Но, разумеется, не в том объеме, как сис-

темному программисту, хакеру или, скажем, разработчику компьютерных игр. Кроме того, в практически любом языке программирования допускаются ***ассемблерные вставки***, многие среды программирования на алгоритмических языках позволяют делать компоновку своих, родных, модулей с модулями на Ассемблере. Это позволяет повысить эффективность программы в наиболее критичных по эффективности местах или реализовать то, что не могут сделать стандартные средства высокоуровневого программирования.

Что можно узнать из этой книги

Эта книга рассчитана в первую очередь на тех, кто *хочет понять принципы обработки информации* на современных ЭВМ: как хранятся в памяти ЭВМ разнообразные данные, как записываются и выполняются арифметические и логические операции, как организуется ввод-вывод информации. **Ассемблер** мы будем изучать постепенно и не сам по себе, а в паре с алгоритмическим языком (**Pascal** или **C++**). Таким образом, мы будем знакомиться с принципами организации и обработки данных в современных компьютерах, изучая не только Ассемблер, но и *углубляя свои знания алгоритмических языков*. Мы научимся читать ассемблерный код, который "выдает" компилятор **C++** (**Borland C++** или **Microsoft Visual C++**). Проанализируем его и, возможно, вам, уважаемые читатели, удастся улучшить ваш исходный код. И в конечном итоге, мы научимся делать свои программы более устойчивыми к внешним воздействиям и поймем, как и зачем нужно делать "защиту от дурака".

Рекомендации по чтению книги

Если вы только начинаете осваивать безбрежный океан Ассемблера, то предполагается, что читать эту книгу вы будете, конечно, сидя за компьютером, что вы немного знаете хотя бы один из алгоритмических языков программирования (**Pascal** или **C/C++**) и основы алгоритмизации (линейные, разветвляющиеся и циклические алгоритмы). Хорошо будет для понимания вами существа дела и приобретения опыта, если вы сделаете и парочку задач из лабораторного практикума. Все лабораторные работы содержат примеры типовых решений. Конечно, вы можете ими воспользоваться, разобраться и сделать решение еще лучше. Ведь улучшать что-то уже сделанное гораздо проще, чем придумывать самому. А не получится сразу улучшить, тоже ничего страшного — просто у вас пока маловато знаний... Со временем количество обязательно перейдет в качество и у вас, уважаемый мой читатель, все получится!

Минимальный набор инструментальных и программных средств

- Компьютер **IBM PC** с **Intel**-совместимым процессором (i86 — если найдете, i286, i386, i486, Pentium,...);
- Операционная система линейки **Windows 9x/Me/XP/NT 4/2000** (на первых порах — окно MS DOS);
- Компиляторы фирмы **Borland: Pascal-7.x + TASM-3.2** (входит в стандартную поставку **Borland Pascal-7.x**); или компиляторы **Borland C++ 3.1 + TASM-3.1** (входит в стандартную поставку **Borland C++ 3.1**); или компиляторы **Borland C++ 4.x, 5.02 + TASM-4.1** (входит в стандартную поставку **TASM-5**).

Вообще говоря, можно взять и другую раскладку компиляторов с языка Ассемблера (например, компиляторы **Watcom** — **WASM** или **Microsoft** — **MASM**), но для начинающих осваивать **Ассемблер**, по мнению автора, лучше остановиться именно на компиляторах фирмы **Borland (TASM)**, как наиболее доступных и сбалансированных в смысле совместной работы разноязычных модулей (**Pascal** и **Assembler**, **C/C++** и **Assembler**). Кроме того, **Turbo Assembler** в действительности содержит в себе два синтаксических стандарта языка Ассемблера — **MASM** и **Ideal**, которые мы будем широко использовать. Освоившись с компиляторами фирмы **Borland**, можно смело "пускаться в плавание" и с другими компиляторами. И мы с вами это тоже периодически будем делать.

На кого рассчитана книга

Книга предназначена для программистов, как начинающих, так и тех, кто хочет *глубже разобраться в особенностях организации и функционирования ЭВМ, чтобы грамотно программировать на любом алгоритмическом языке и отлаживать реальные задачи и действительно понимать, как работает ЭВМ.*

По мнению автора, книга может быть также полезна преподавателям и студентам, профессионально изучающим программирование.

Что есть в книге

Материал книги базируется на лекционном курсе и лабораторном практикуме "Основы организации и функционирования ЭВМ". Поэтому книга состоит из двух частей: лекции и лабораторные работы. Автор отлаживала эту методику преподавания, начиная с 1976 года, пропустив через себя и студентов Национального аэрокосмического университета имени Н.Е. Жуковского (ХАИ) 5 разных ассемблеров совместно с 20 языками программирования.

Книга снабжена большим количеством примеров и готовых программ как на языке Ассемблера (TASM-5, MASM-6.13), так и на алгоритмических языках Pascal (Borland Pascal-7.0, Delphi-5), C/C++ (Borland C++ 3.1, 4.5, 5.02, Borland C++ Builder 5, Visual C++ 6.0). Большая часть этих программ была разработана еще в операционной системе MS DOS (в соответствующих времени программных средах), затем при подготовке рукописи книги к изданию дополнена, проверена и откомпилирована в операционных системах **Windows NT 4 Server (SP6postfix)/Windows 2000 Professional SP2.** Все эти программы содержатся на прилагаемой к книге дискете и на сайте издательства.

В конце книги дан перечень использованной и рекомендуемой литературы, а также полезных электронных ресурсов, которые становятся жизненно необходимыми в наш век быстрой смены всего и вся.

Как связаться с автором

Узнать больше о программировании на C/C++ можно на сайте автора книги

http://www.anriintern.com/computer/c++/ — "Язык C++ для начинающих",

который входит в состав портала бесплатного дистанционного образования "Anri Education Systems" (http://www.anriintern.com/). Возможно, со временем я организую нечто подобное и для осваивающих Ассемблер, если в этом будет необходимость.

Ваши вопросы, замечания и пожелания по данной книге и смежным вопросам будут с благодарностью приняты автором по адресу: **n_golub@ukr.net** (просьба в теме письма (Subject) указывать "Assembler").

Благодарности

Я благодарна всем своим студентам и ассистентам (тоже бывшим моим студентам), совместно с которыми уточнялись постановки задач и шлифовались предлагаемые вам в этой книге примеры — их фамилии "увековечены" в листингах: **Лучшев Павел Александрович, Левин Сергей (младший), Астафьев Артем, Патлань Игорь, Устименко Вадим** и многие-многие другие, чьи опусы я храню с 1990 г.

Конечно, я благодарна заведующему кафедрой программного обеспечения ХАИ, моему учителю и наставнику, профессору **Сироджe Игорю Борисовичу**, доверившему мне (в то время достаточно сформировавшемуся программисту-самоучке) это не простое дело — преподавание программирования вообще и Ассемблера в частности.

Особая благодарность начальнику вычислительного центра ХАИ, а ныне декану инженерно-менеджерского факультета, профессору **Кожухову Валерию Дмитриевичу**, не побоявшемуся принять меня, только что получившей диплом инженера-механика (после шести месяцев дипломирования на Вычислительном центре Сибирского отделения Академии Наук СССР. г. Новосибирск), на должность программиста и доверившему установить, как потом выяснилось — впервые в городе, альфа-транслятор. Именно потребность отвечать на множество вопросов неискушенных и очень даже искушенных программистов вынудила меня, инженера-механика, разобраться, как же там все работает "внутри" и почему программа иногда работает, а иногда нет или вообще работает неверно. Так я самостоятельно пришла к Ассемблеру из Алгола-60 (его подмножества — альфа) — был в то время даже такой термин "Туманности Алгола".

Спасибо моим друзьям-коллегам за их постоянную готовность помочь и доброжелательную критику: **Левину Сергею Владленовичу (старшему), Корниенко Олегу Григорьевичу**, а также их гостеприимным и хлебосольным семьям.

Спасибо за помощь, понимание и постоянную поддержку моим замечательным и близким друзьям, ныне проживающим за границей: **Маше Соколовой** (Лондон, Великобритания), **Ларисе Итиной** (Бостон, США).

Огромная моя благодарность тем, кто делал вместе со мной эту книгу, — всему прекрасному, дружному и доброжелательному коллективу издательства "ДиаСофт" и особенно ее региональному представителю **Абрамовичу Виктору Уриэльевичу.**

Искренняя моя благодарность всем моим родным и близким, которые с пониманием и бесконечным терпением относились и относятся к моим ночным бдениям за экраном персонального компьютера — такова уж доля программиста: поймать, а иногда даже и подержать за хвост постоянно ускользающую "птицу счастья".

И, наконец, спасибо тебе, ласковый (иногда кусачий), теплый, пушистый комочек — очень серьезный и знающий, постоянно сидящий со мной за компьютером вот уже пять лет сиамский кот **Байт Пинкертон (Пиня)**.

Лекции

Мы тогда достигаем, когда не можем, но делаем. Ибо не можем часто относительно, нам лишь кажется, что не можем.

Е.И.Рерих (1879–1955гг.)

Любая программа оперирует данными. В простейшем случае данные — это знакомые нам целые и вещественные числа. В более сложном — это сами программы. Знакомиться с Ассемблером мы будем постепенно, используя аналогии с алгоритмическими языками, иногда забегая немного вперед (опережающая ссылка), но в целом сохраняя принцип от простого к сложному.

1

Позиционные системы счисления

Будь благословенно божественное число, породившее богов и людей.

Пифагор Самосский (VI век до н.э.)

Познакомимся сначала с *общими принципами организации данных* в компьютерах. Для этого нам надо разобраться в *системах счисления* и научиться считать в основных машинных системах счисления: двоичной и шестнадцатеричной. *Это ОЧЕНЬ важно для ГРАМОТНОГО программирования на любом алгоритмическом языке.*

Люди с древнейших времен пытались установить и усовершенствовать систему счисления. До наших времен дошли, например, такие известные системы счисления, как римская и арабская. Чем же они отличаются? Оказывается, с точки зрения современной вычислительной техники, — одна из них является *позиционной* системой счисления, а другая нет. Какая же? И что значит *позиционная система счисления*? Оказывается, это такая система счисления, где позиция цифры однозначно связана со значением числа и, зная номер позиции и систему счисления, можно получить достаточно простую формулу для вычисления значения всего числа. Попробуем разобраться в этом с формальной точки зрения.

Пусть задано вещественное число, состоящее из **n+1** цифр целых (**Ц**) и **k** цифр дробных (**Д**). Пронумеруем, начиная с нуля, позиции числа от десятичной точки вправо и влево. В этом случае позиционной системой счисления с основанием **q** является такая система, в которой каждая цифра числа ($Ц_i$ или $Д_i$) принадлежит множеству {0,1,...,**q**-1}, а числовая позиция **i = [-k, n]** имеет свой, строго определенный смысл (см. формулу ниже). Теперь представим наше число с позиционной точки зрения в следующем виде:

число	$Ц_n$	$Ц_{n-1}$	...	$Ц_1$	$Ц_0$	.	$Д_{-1}$	...	$Д_{-k}$
позиция	n	n-1	...	1	0		-1	...	-k

$$Ц_n \cdot q^n + Ц_{n-1} \cdot q^{n-1} + \ldots + Ц_1 \cdot q^1 + Ц_0 \cdot q^0 + Д_{-1} \cdot q^{-1} + \ldots + Д_{-k} \cdot q^{-k}$$

Забегая немного вперед, скажем, что с помощью этой формулы можно перевести число из любой позиционной системы счисления с основанием **q** в десятичную систему. Этот принцип широко используется во всех современных ЭВМ при переводе числа из любимой машинной двоичной позиционной системы счисления в десятичную (см. п. 1.2.4). Говорят, что стандартный метод записи числа в позиционной системе счисления с основанием **q** основан на степенях **q**.

Как известно, есть числа целые и вещественные (правильные дроби и смешанные числа). Целые и вещественные числа могут быть положительными или отрицательными. Из элементарной математики известно, что вещественное число, в котором числитель меньше знаменателя, называется *правильной дробью*. Вещественное число, содержащее целую и дробную части, называется *смешанным*. Дробную часть будем называть *мантиссой*.

Существует **общее правило** перевода чисел из десятичной системы в любую другую позиционную систему счисления с основанием **q**.

Для перевода ***целого десятичного числа*** в систему счисления с основанием **q** необходимо данное число последовательно *делить* на это основание до получения целого остатка, меньшего, чем **q**. Результат в системе счисления с основанием **q** представится в *виде упорядоченной последовательности остатков от деления в порядке, обратном их получению* (старшую цифру числа дает последний остаток, а младшую — первый), — подробнее см. п. 1.2.1.

Для перевода *вещественного десятичного числа (**правильной дроби**)* в систему счисления с основанием **q** необходимо данное число последовательно *умножать* на это основание до тех пор, пока в мантиссе не получится либо чистый нуль (что в общем случае достаточно проблематично), либо нужное количество разрядов. Результат в системе счисления с основанием **q** представится в *виде упорядоченной последовательности целых чисел* (мантисса при этом отбрасывается), — подробнее см. п. 1.2.2.

Вещественное смешанное число в системе счисления с основанием **q** образуется, как и в десятичной, путем соединения (*конкатенации*) целой и дробной частей, полученных отдельно в системе счисления с основанием **q** — см. п. 1.2.3.

1.1. Десятичная система счисления (Decimal)

Арабская система счисления является позиционной, а римская — нет. Проверим это на примерах. Число в арабской десятичной системе счисления (**q=10**) состоит из множества цифр {**0,1,2,3,4,5,6,7,8,9**}.

число	5	3	4	2
позиция	3	2	1	0

$$5 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0 = 5000 + 300 + 40 + 2 = 5342$$

число	8	2	5	6	3	.	9	4	7
позиция	4	3	2	1	0		-1	-2	-3

$$82563.947 = 8*10^4 + 2*10^3 + 5*10^2 + 6*10^1 + 3*10^0 + 9*10^{-1} + 4*10^{-2} + 7*10^{-3} =$$
$$80000 + 2000 + 500 + 60 + 3 + 0.9 + 0.04 + 0.007$$

Все привычные вычисления в десятичной системе счисления, оказывается, точно так же работают и во всех других системах счисления, если они позиционные. А как мы увидим в дальнейшем, все известные машинные системы счисления являются позиционными.

Теперь возьмем любое римское число: XIV, LI, IV... Мы видим, что таких закономерностей в общем случае не наблюдается. Зато римские числа прекрасно смотрятся, например, на циферблатах часов (каждому свое).

1.2. Двоичная система счисления (Binary)

Итак, перейдем к конкретной "любимой" машинной системе счисления — двоичной (**bin**). Ее основание q=2. Число в этой системе счисления образовано из множества цифр {0,1}. Базовая единица компьютерных данных называется **бит**. *Слово **БИТ** является удобным сокращением неуклюжего английского выражения **binary digit**, т.е. двоичная цифра*, — П. Нортон [Л4-9]. Таким образом, *двоичное число — это последовательность бит.*

1.2.1. Перевод целых десятичных чисел в двоичную систему счисления

Для перевода ***целого десятичного числа*** в систему счисления с основанием q=2 необходимо данное число последовательно *делить* на число **2** до получения целого остатка, меньшего, чем **2**. Результат в двоичной системе счисления (сокращенно **bin**) *будет иметь вид упорядоченной последовательности остатков от деления в порядке, обратном их получению* (старшую цифру числа дает последний остаток, а младшую — первый). Разберемся с этим на конкретных примерах.

ПРИМЕР 1.1А.

Переведем из десятичной в двоичную систему счисления (СС) число 137:
137 → ??? bin

Не забудьте, что при получении результата **остатки** нужно взять в **ОБРАТНОМ порядке**!

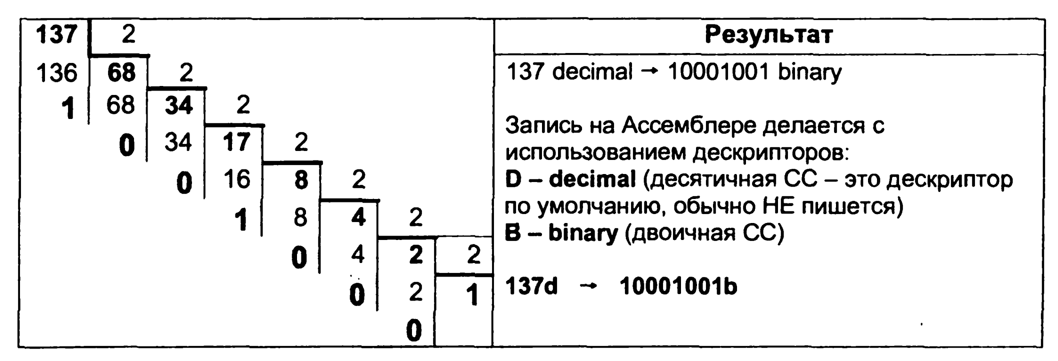


РИС.1.1. *Схема перевода для небольших целых чисел (Decimal → Binary).*



ПРИМЕР 1.2А.

Переведем из десятичной в двоичную систему счисления (СС) число 13789:

13789d → ??? bin

Выполнив аналогичное предыдущему деление, получим следующий результат:

13789d → 11010111011101b

Проверьте себя сами. Получился ли тот же результат? Этот результат можно получить и по-другому — см. Пример 1.2b.

1.2.2. Перевод правильных десятичных дробей в двоичную систему счисления

Для перевода *вещественного десятичного числа (**правильной дроби**)* в систему счисления с основанием **q=2** необходимо данное число (**только мантиссу**) последовательно *умножать* на число **2** до тех пор, пока в мантиссе не получится либо чистый нуль, либо нужное количество разрядов. При умножении получаемые целые значения НЕ учитываются. Зато они последовательно засчитываются в результат. Получается *упорядоченная последовательность целых двоичных чисел.* Опять проследим этот алгоритм на конкретных примерах.

ПРИМЕР 1.3А.

0.5d → ??? bin Здесь результат получается сразу: 0.5d → 0.1b

	0.5
x	2
1	*.0*



ПРИМЕР 1.4А.

0.703125d → ??? bin

	0.703125
x	2
1	. 406250
x	2
0	. 812500
x	2
1	. 625000
x	2
1	. 250000
x	2
0	. 500000
x	2
1	. *000000*

В данном примере нам очень повезло — мы получили мантиссу, равную нулю. Соберем последовательно **сверху вниз** все целые части нашей дроби, получим результат:

0.703125d → 0.101101b

ПРИМЕР 1.5А.

0.05d → ??? bin

```
      0.05
x        2
----------
0      .10
x        2
----------
0     . 20
x        2
----------
0     . 40
x        2
----------
0     . 80
x        2
----------
1     . 60
x        2
----------
1     . 20
x        2
----------
0     . 40
```

....

Мы зациклились. Получается двоичная ПЕРИОДИЧЕСКАЯ дробь:

0.05d → 0.000011(0011)b

Опять нам повезло, мы не очень долго умножали... Но такое бывает достаточно редко.

На этом примере мы столкнулись с ОЧЕНЬ ВАЖНЫМ фактом при программировании: **вещественные данные в общем случае НЕВОЗМОЖНО точно представить в памяти ЭВМ**. Точность представления (количество точно представляемых десятичных разрядов вещественного числа) зависит от разрядности компьютера. Более подробно мы это рассмотрим в п.2.3.

1.2.3. Перевод смешанных десятичных чисел в двоичную систему счисления

Вещественное смешанное число в системе счисления с основанием **q=2** образуется, как и в десятичной, путем соединения (*конкатенации*) целой и дробной частей.


ПРИМЕР 1.6А.

117.25d → 1110101.01b

Проверьте себя сами. Нужно целую и дробную часть переводить отдельно так, как мы это уже делали. Получился ли тот же результат? Если получился, очень хорошо. Если же нет, не расстраивайтесь, а просто вернитесь на шаг или два назад. И не забывайте, что двоичная система счисления — любимая система счисления для компьютера, но отнюдь не для человека. Постарайтесь понять компьютер...

1.2.4. Перевод двоичных чисел в десятичную систему счисления

Двоичная система счисления является позиционной системой счисления с основанием **q=2**, поэтому для нее справедлива формула, с которой мы познакомились в начале главы 1. Т.е. мы можем разложить двоичное число по степеням двойки. Проверим правильно ли мы решили пример 1.6а.



ПРИМЕР 1.6А.

1110101.01b → ??? dec

число	1	1	1	0	1	0	1	.	0	1
позиция	6	5	4	3	2	1	0		-1	-2

```
1*2^6 + 1*2^5 + 1*2^4 + 1*2^2 + 1*2^0 + 1*2^-2 =
64 + 32 + 6 + 4 + 1 + 1/4 = 117.25
```

Итак, получили результат:

1110101.01b → 117.25d

Пример 1.6а решен правильно! Степени числа 2 представлены в Прил. 1 (см. табл.П1.1).

1.3. Шестнадцатеричная система счисления (Hexadecimal)

Если вы хотите понять тонкости работы любого персонального компьютера, Вам необходимо иметь достаточное представление о специальной компьютерной системе счисления, которая называется шестнадцатеричной... Это разумный компромисс между тем, что удобно компьютеру, и тем, что удобно человеку, — П. Нортон [Л4-9].

Эта система счисления (**hex**) определена на множестве цифр **{0,1,2,3,4,5,6,7,8,9, a,b,c,d,e,f}** и позволяет существенно сжать двоичную информацию, облегчая тем самым ее выдачу в разнообразных листингах. А человек, анализируя данную информацию, может при необходимости сам получить двоичную информацию и в ней разбираться (этим мы с Вами тоже очень скоро займемся). Связь между этими системами счисления достаточно простая — одна шестнадцатеричная цифра соответствует четырем (*тетраде*) двоичным так, как показано в Прил.1 (см. табл. П1.2). Шестнадцатеричные цифры {A,B,C,D,E,F} в Ассемблере НЕ чувствительны к регистру, поэтому мы их будем писать то малыми буквами, то большими, руководствуясь наглядностью их представления.

Шестнадцатеричная система счисления — это базовая система при общении компьютера с человеком вообще и в операционной системе MS DOS и Windows в частности. Поэтому с ней мы познакомимся более подробно.

1.3.1. Перевод десятичных чисел в шестнадцатеричную систему счисления

Чтобы перевести число из десятичной СС в шестнадцатеричную, можно воспользоваться общим правилом, т.е. последовательно делить число на 16, пока не получим остаток меньше, чем 16 (для целых чисел), или умножать на 16 (для правильных дробей). Это не совсем удобно, при делении или умножении легко ошибиться. Поэтому принято сначала

перевести число в двоичную СС так, как мы это делали в примере 1.1а или в примерах 1.3а-1.5а, а потом каждую двоичную тетраду представить шестнадцатеричной цифрой — см. табл. П1.2. Тетрады нужно получать, **начиная от десятичной точки** (влево — для целой части числа, вправо — для дробной), добавляя, если нужно, **незначащие нули**. Посмотрим, как это делается, на примерах, воспользовавшись уже сделанным переводом из десятичной в двоичную систему счисления в предыдущих примерах 1.1а-1.6.



ПРИМЕР 1.1В.

Переведем число 137 из десятичной в шестнадцатеричную СС, используя двоичную систему счисления как промежуточную:

137 → ??? hex

137d → 1000 1001b

8 **9h**

137d → **89h**

ПРИМЕР 1.3В.

0.5d → ??? hex

0.5d → 0.1b = 0.1*000*b

0. 8h

В этом примере мы, для того чтобы получить тетраду, двоичное число *расширили незначащими нулями* (точно так, как это делается и в обычной десятичной системе счисления). И получили результат:

0.5d → 0. 8h

ПРИМЕР 1.4В.

0.703125d → ??? hex

0.703125d → 0.1011 01b = 0.1011 01*00*b

0. B 4h

Здесь тоже нужно сделать расширение до двоичной тетрады:

0.703125d → 0.B4h

ПРИМЕР 1.5В.

0.05d → ??? hex

0.05d → 0.000011(0011)b = 0.0000 1100 11*00* *(1100)*b

0. 0 C C (C)h

В этом, более сложном примере, мы расписали периодическую дробь так, чтобы получить нужные двоичные тетрады. И опять имеем периодическую, на этот раз шестнадцатеричную, дробь:

0.05d → 0.0C(C)h

ПРИМЕР 1.6В.

117.25d → ??? hex

117.25d → 111 0101.01b = *0*111 0101. 01*00*b

7 5. 4h

В настоящем примере нам пришлось расширить до тетрады и дробную, и целую части двоичного числа. В результате получаем:

117.25d → 75.4h

Для перевода достаточно *больших целых* десятичных чисел, например, 13789, операция деления на два достаточно трудоемкая (а на 16 еще того хуже!). Можно воспользоваться таблицей П1.3 и сразу перевести число в шестнадцатеричную систему счисления.

ПРИМЕР 1.2В.

13789d → ??? hex

Для этого нужно найти в табл. П1.3 ближайшее десятичное число, меньшее или равное исходному, и его шестнадцатеричный эквивалент:

13 789d → 12 288d → **3000h**

затем вычесть найденное число из исходного и операцию повторить:

13 789				
− 12 288				
1 501d	→	1 280d	→	**500h**
− 1 280				
221d	→	208d	→	**D0h**
− 208				
13d	→			**Dh**

Далее, поскольку шестнадцатеричная система счисления тоже позиционная, складываем полученные числа. Итого, получаем:

13 789d → 35DDh.

Теперь, если нужно, мы можем сделать перевод из шестнадцатеричной системы счисления в двоичную, воспользовавшись табл. П1.2:

13 789d → 3 5 . D Dh.

*00*11 0101 1101 1101b

Незначащие (*ведущие*) нули можно удалить. Итого наш результат имеет вид:

13 789d → 35DDh → 11 0101 1101 1101b.

1.3.2. Перевод целых шестнадцатеричных чисел в десятичную систему счисления

Поскольку между шестнадцатеричной и двоичной системами счисления существует очень тесная связь (см. табл. П1.2), обычно переводят *любое* шестнадцатеричное число в десятичное в два этапа, используя следующую схему:

hex → bin → dec.

Для *целых* чисел можно сделать это сразу, используя табл. П1.3.

ПРИМЕР 1.2С

35DDh	→	??? dec
3000h	→	12 288
500h	→	280
D0h	→	208
Dh	→	13

Опять все складываем, получаем:

35DDh → 13 789d.

И, наконец, можно переводы из одной системы счисления в другую (Dec, Bin, Oct, Hex) делать и через стандартный Windows-калькулятор (режим Scientific — научный), но только для **ЦЕЛЫХ** чисел. Числа дробные пока, к сожалению, нужно переводить вручную...

1.3.3. Биты, байты, полубайты, слова и двойные слова

Информация в любом компьютере хранится в определенных ячейках памяти. Для базовой конфигурации **IBM PC XT/AT** ячейки для хранения целочисленных данных имеют длину байт, слово и двойное слово.

Мы уже знаем, что *бит* (**bit**) — это двоичный разряд **{0,1}**. Последовательность смежных двоичных цифр длиною в 8 бит получила название *байт* (**byte**). Байт — это наименьшая адресуемая компьютером единица информации.

Также мы знаем, что шестнадцатеричное число — это последовательность смежных четырех бит (тетрада). Тогда байт содержит в себе два шестнадцатеричных числа, т.е. иногда еще говорят, что байт состоит их двух полубайтов. Таким образом, максимально возможное число, которое размещается в байте, в двоичном формате будет состоять из восьми единиц, а в шестнадцатеричном — из двух цифр FF:

FFh → 1111 1111b → 255d

Всем ли понятно, как мы это вычислили? Надо просто перевести число из шестнадцатеричной (двоичной) системы счисления в десятичную, пользуясь теми приемами, которые мы уже изучили.

Слово (**word**) — это упорядоченная последовательность информации длиной в 2 байта. Таким образом, вычисляем, какое максимально возможное целое значение можно поместить в слово:

FFFFh → 1111 1111 1111 1111b → 65535d

Двойное слово (**double word**) — это упорядоченная последовательность информации длиной в 2 слова (4 байта). Какое же максимально возможное целое число можно разместить в двойном слове?

FFFF FFFFh → **1111 1111 1111 1111 1111 1111 1111 1111b** (см. Табл. П1.3)

F000 0000h	→	4026531840d
F00 0000h	→	251658240d
F0 0000h	→	15728640d
F 0000h	→	983040d
F000h	→	61440d
F00h	→	3840d
F0h	→	240d
Fh	→	15d
		4294967295d

Или воспользуемся табл.П1.1 следующим образом:

1111 1111 1111 1111 1111 1111 1111 1111b → **2^{32} – 1=2 * 2^{31} – 1 =**

2 * 2147483648 – 1 = **4294967295**

Так, зная длину ячейки, можно сориентироваться, какое максимально возможное число может в ней поместиться. **Понимание этого момента ОЧЕНЬ важно при программировании на любом алгоритмическом языке, в любой операционной системе и на любой компьютерной архитектуре!**

Теперь вы сумеете вычислить сами, какое максимально возможное число может разместиться в 64-х битах, 128 битах и т.д. А это уже персональные компьютеры последнего поколения...

1.4. Восьмеричная система счисления (Octal)

Исторически сложилось так, что восьмеричная система счисления появилась немного раньше шестнадцатеричной. В настоящее время широко используется в операционных системах **Unix**. По своим идеям и принципам она полностью идентична шестнадцатеричной системе счисления, но сжимает *триаду* (тройку) бинарных разрядов. Эта система счисления (сокращенно **oct**) определена на множестве цифр **{0,1,2,3,4,5,6,7}**.

ПРИМЕР 1.6С.

117.25d → ??? oct

117.25d → 111 0101.01b = *00*1 110 101. 01*0*b

1 6 5. 2o

В этом примере нам тоже пришлось расширить до триады и дробную, и целую части двоичного числа, как в примере 1.6b. В результате получаем:

117.25d → 165.2o

Как будет выглядеть максимально возможная целочисленная величина, которую можно разместить в байте, в восьмеричном виде? Попробуйте решить эту задачку сами... А решив ее, вы увидите, что и работать с шестнадцатеричной системой проще и удобнее.

2

Формат представления базовых данных в IBM PC

Овладей предметом, а слова найдутся.

Квинт Гораций Флакк (65-8 гг. до н.э.)

Персональный компьютер (ПК) оперирует с большим количеством самых разнообразных данных, имеющих определенный *формат*, определяемый размером ячейки (количество бит), где данное хранится, и способом его представления. Форматы допустимых данных зависят от модели персонального компьютера и от набора команд для их обработки. Для начала мы разберемся с **базовым** набором данных и команд — набором для IBM PC XT (процессоры **Intel8088/8086**). Этот набор принят по умолчанию, независимо от модели ПК. Набор форматов данных для моделей **i286** от базового ничем НЕ отличается, добавлены только несколько команд. Этот базовый набор образует так называемую платформу **Win16,** обеспечивающую 16-разрядное программирование. Поняв базовый принцип организации данных, мы легко поймем и дальнейшие расширения форматов данных и команд для **i386, i486, i586** и совместимых с ними моделей ПК, обеспечивающих работу в **Win32**.

Любая величина, видимая программе, имеет тип и должна быть соответствующим образом представлена и описана. Займемся для начала **простыми** типами данных: символы, целые и вещественные числа. В алгоритмических языках, таких, как Pascal или C/C++, они описываются с помощью соответствующих ключевых слов (в тексте нашего учебного курса и в редакторе современных компиляторов они, как правило, выделяются **жирным** шрифтом).

Как правило, в задаче всегда известен *диапазон* вводимых данных (*область определения*). Диапазон результирующих данных известен далеко не всегда. Поэтому обычно считают, что такие данные должны иметь максимально допустимый диапазон значений. При этом нужно учитывать, что *данные более длинные требуют и больше памяти* — см. табл.П4.1-П4.2.

2.1. Символы

Символы (**characters**) в компьютере хранятся в виде числового кода. В США, естественно, наиболее распространенным является американский стандартный код для обмена информацией (*American Standard Code for Information Interchange* — **ASCII**). Это основной 7-битовый двоичный код, в котором представляются алфавитные, цифровые и специальные символы. Символы могут быть **управляющие** (коды **0..31, 127**) и **видимые** на экране дисплея или на бумаге при печати (коды **32..126**) — из них образуются строки символов. В Приложении 2 (табл. П2.1 и П2.2) показаны все эти коды и соответствующие им символы.

Нетрудно заметить, что для представления символов достаточно одного байта. Но в байте содержатся 8 бит. Остается незадействованной добрая половина кодовой таблицы. Поэтому коды **128..255** были выделены для так называемых **национальных символов и алфавитов, а также для символов псевдографики**. Например, для кириллицы в операционной системе MS DOS довольно долгое время существовали три разные кодовые таблицы, пока не был принят единый стандарт — остановились на так называемой *альтернативной кодировке* — см. табл. П2.3.

Теперь, в рамках Windows, появился новый стандарт кодирования символов — Windows ANSI, который тоже является однобайтовой схемой кодирования. Первые символы 0..127 соответствуют кодировке ASCII, вторая половина — опять национальные алфавиты. Соответственно, для кириллицы появилась тоже другая кодировка (**Win**) — кодовая страница (**Code Page 1251**) — ср-251 — см. табл. П2.4. А есть еще кодировки кириллицы MAC, ISO, KOI-8r (под Unix) [Л4-7] ...

А что делать тем, у кого, например, иероглифы? Как им работать с компьютером? Был придуман **Unicode** — стандарт кодировки символов, имеющий фиксированную длину представления одного символа (16 бит, или 2 байта) и позволяющий закодировать все алфавиты в мире. Это — родной кодовый набор для Windows NT [Л3-1]. Поэтому иногда со шрифтами кириллицы у пользователей Windows NT возникают проблемы...

Есть еще так называемые **расширенные ASCII-коды** (в литературе и в переводной электронной документации их иногда путают с кодами 128..255, которые тоже являются расширением (extended) ASCII). Это всегда последовательность двух символов по одному байту: в первом находится ноль, во втором — так называемый **scan**-код. Перечень этих кодов приведен в Приложении 3 (см. табл. П3.1, П3.2).

При выполнении наших примеров мы с Вами будем работать с символами кириллицы в кодировке DOS (ср-866) или Win (ср-251).

2.2. Целые числа

Если вы уже знакомы с каким-либо алгоритмическим языком (что ОЧЕНЬ желательно), то вы знаете, что, например, в **C/C++** целые величины могут иметь тип **char** или **int**. Каждое из этих данных может быть ***знаковое*** (т.е. положительное или отрицательное) — **signed** или ***беззнаковое*** (*только положительное!*) — **unsigned**, а тип **int** еще может быть коротким — **short** или длинным — **long** — см. табл.П4.1 или табл. П4.2. Все эти типы данных имеют определенную длину ячейки, а отсюда и допустимый диапазон значений.

Если Вам ближе алгоритмический язык **Pascal**, то идеологически здесь все то же самое, только описываются данные немного по-другому (см. табл. П4.3 или табл. П4.4).

В Ассемблере программист сам решает, какие данные (целые или вещественные) он поместит в ту или иную ячейку. Например, **простое арифметическое** данное, под которое отведено 32 или 64 бита, может быть как целым, так и вещественным — см. табл. П4.2 или табл. П4.4. Поэтому продвинутые программисты часто говорят о коротких или длинных целых, а также о коротких или длинных вещественных числах.

Разберемся вначале с целочисленными данными и погрузимся в детали, которые вначале так пугают... А освоившись с ними, программист поймет, как же работает компьютер и как помочь ему (компьютеру) ПРАВИЛЬНО решить ту или иную задачу.

2.2.1. Целые числа без знака

Как видно из области допустимых значений для платформы **Win16** (см. табл.П4.1 или П4.3), целые числа без знака — это ПОЛОЖИТЕЛЬНЫЕ числа (или НОЛЬ) и они могут занимать 8, 16 или 32 бита памяти ЭВМ. Считается, что бит 0 — младший бит (для удобства будем считать, что он расположен крайним СПРАВА). Старший бит — 7 (15 или 31) — для удобства расположим крайним СЛЕВА. Для БЕЗЗНАКОВЫХ данных **все** биты считаются информационными.

Информационное поле числа {0,1}																		
31	...	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
старший	Биты																	младший

РИС. 2.1. *Формат представления для 32-битного БЕЗЗНАКОВОГО числа.*

 **ПРИМЕР 2.1.**

Пусть имеется десятичное целое число 40000. В каком формате его можно представить? Т.е. сколько ему требуется бит и каково будет его внутреннее (машинное) представление?

Решение. Для начала нужно перевести это число в двоичную систему счисления. Для этого можно воспользоваться табл.П1.3 или стандартным калькулятором Windows (режим Scientific — научный). Остановимся на калькуляторе. Получаем:

40000d → 1001 1100 0100 0000b → 9C40h

По количеству полученных бит видно, что для хранения такого числа нам понадобится *минимум 16 двоичных разрядов.* В терминах алгоритмического языка C/C++ это формат **unsigned int** (см. табл.П4.1) для платформы **Win16**, а для Pascal'я — это формат **word**.

Можно это число (*с запасом*) разместить и в 32-х битах:

40000d → 0000 0000 0000 0000 1001 1100 0100 0000b → 0000 9C40h

Обратите внимание на *ведущие* НУЛИ!!!

Это будут для C/C++ соответственно, форматы: **unsigned long, long** (см. табл.П4.1 — Win16) или **unsigned int, int, unsigned long, long** (см. табл.П4.2 — Win32). Для Pascal'я — это форматы: **LongInt** (см. табл.П4.3 — Win16) или **Integer, LongInt, LongWord, Cardinal** (см. табл.П4.4 — Win32).

Мы с Вами в п.1.3.3 определили, какое же максимальное двоичное, шестнадцатеричное и десятичное значение можем разместить, имея в распоряжении 8, 16 или 32 бита. Отсюда становится понятно, почему у этих данных такая область допустимых значений — больше информации в отведенную ячейку памяти просто НЕ поместится.

Надеюсь, что теперь вы сами сможете определить, какой диапазон допустимых значений может дать, например, целое беззнаковое данное длиной в 64 бита (см. табл.П4.4). А это уже платформа **Win32**!

2.2.2. Целые числа со знаком

Внимательно разглядывая табл. П4.1-П4.2, можно увидеть, что целые ЗНАКОВЫЕ данные, во-первых, могут быть как ПОЛОЖИТЕЛЬНЫМИ, так и ОТРИЦАТЕЛЬНЫМИ, включая НОЛЬ, и, во-вторых, они по диапазону допустимых значений в два раза меньше беззнаковых (для положительных значений). Почему?

Это происходит, потому что НЕ вся область бит отводится под информацию, как было с беззнаковыми данными. **Старший бит всегда отводится под знак**. Он называется бит **S** — **Signum** (знак — лат.):

S = **0** — для ПОЛОЖИТЕЛЬНЫХ чисел;

S = **1** — для ОТРИЦАТЕЛЬНЫХ чисел.

S	Информационное поле числа {0,1}																	
31	...	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Знак	Биты																	

РИС. 2.2. *Формат представления для 32-битного знакового числа.*

Поэтому под информацию отводится всегда на ОДИН бит МЕНЬШЕ.

А теперь немного отвлечемся на небольшую задачку: что получится, если мы исходное десятичное число 12345 лишим последней цифры 5? Получим число 1234. В программировании это называется *сдвиг вправо на один десятичный разряд*. И наше число получилось в *10* раз меньше! Совершенно аналогично происходит и с двоичным числом, если мы его сдвинем на один, но **двоичный**, разряд вправо. Исходное число уменьшится в **2** раза! Попробуйте сделать это сами, например, было число 111001101b, а стало11100110b. Действительно ли оно уменьшилось в два раза? Забегая немного вперед, можно сказать, что *сдвиги* — ОЧЕНЬ важная деталь в программировании вообще, и в Ассемблере в частности.

Если вы поняли эту маленькую задачку, значит, разобрались, почему максимальный диапазон для ПОЛОЖИТЕЛЬНЫХ знаковых чисел получается в два раза меньше. А если НЕ поняли, нужно перейти к схеме, описанной в п.1.3.3. Например, для ПОЛОЖИТЕЛЬНОГО **знакового** данного, которому отведен один байт информации, получается следующее:

0111 1111b → 7Fh → 127d

А как быть с **ОТРИЦАТЕЛЬНЫМИ** числами? Здесь может быть НЕСКОЛЬКО вариантов решений. Ребята из фирмы IBM решили взять за основу представление отрицательных чисел в **ДОПОЛНИТЕЛЬНОМ** коде, суть которого заключается в следующем:

1) Берем МОДУЛЬ отрицательного числа (т.е. число положительное) и по известным Вам алгоритмам переводим **dec** → **hex;**

2) Делаем *инверсию* **hex**-кода (т.е. нули становятся единицами, а единицы — нулями);

3) К полученному **hex**-коду **добавляем 1**.

ПРИМЕР 2.2.

Пусть имеется десятичное число -1607. В каком формате его можно представить? Т.е. сколько ему требуется бит и каково будет его внутреннее (машинное) представление?

Решение. Наше число ОТРИЦАТЕЛЬНОЕ, значит, его нужно представить в ДОПОЛНИТЕЛЬНОМ коде:

1) |-1607| = 1607. Воспользуемся табл. П1.3.

1536	→	600h
71		
64	→	40h
7	→	7h

Таким образом, получаем число **647h** → **0110 0100 0111b.**

Т.е. нашему числу достаточно 12 бит. Но такого формата НЕТ, поэтому берем, например, 16 бит, добавляя ведущие нули: **<u>0000</u>** 0110 0100 0111b

2)Делаем инверсию нашего кода:	**<u>0000</u>**	0110	0100	0111b
Получаем:	**1111**	**1001**	**1011**	**1000b**
3) К полученному **hex**-коду **добавляем**				**1**
Получаем:	**1111**	**1001**	**1011**	**1001b**

Это и есть машинное представление нашего числа -1607 или в шестнадцатеричной системе счисления: **F9B9h.** В памяти компьютера это число будет храниться задом-наперед, т.е. **B9F9h.** Компиляторы MASM версии 5.x и ниже дают именно такой формат. И если такая запись удобна компьютеру, для человека она ОЧЕНЬ неудобна. Поэтому в листингах компилятора TASM (Turbo Assembler) мы получим результат тоже в *удобном для человека* виде: **F9B9h.** Все дальнейшие выкладки и рассуждения мы и будем делать так, как удобно пользователю. А если потребуется истинный, т.е. перевернутый вид, мы к нему еще вернемся...


ПРИМЕР 2.3.

А какой получится машинный результат, если мы захотим разместить наше число -1607 в 32 битах?

Ответ: FFFF F9B9h

Обратите внимание на *ведущие двоичные* ЕДИНИЦЫ!!! Число ОБЯЗАТЕЛЬНО должно быть в ЗНАКОВОМ формате (например, с точки зрения C/C++ это формат **long int** — 32 бита). А если оно будет интерпретироваться как БЕЗЗНАКОВОЕ, получим следующее десятичное значение (обратимся за помощью к Windows-калькулятору):

FFFF F9B9h → **4294965689d**

Здесь мы вышли на еще одно ОЧЕНЬ важное обстоятельство: **двоичный код может быть один и тот же для РАЗНЫХ чисел!!!** Все зависит от программиста, как он этот код интерпретирует... Поэтому для Ассемблера НЕТ директив описания знаковых и беззнаковых данных — см. табл.2.2 (п. 2.4.1). Правда, есть команды, учитывающие данное обстоятельство, но речь об этом впереди...

Понимание этого НЕПРОСТОГО для начинающего программиста момента дает КЛЮЧ для написания ГРАМОТНЫХ программ на любом языке программирования и в разных операционных системах.

А если это же число -1607 — разместить в 64 битах?

Ответ: FFFF FFFF FFFF F9B9h

Если вы НЕ поняли, как мы получили такие два результата, прочтите еще раз АЛГОРИТМ получения отрицательного числа в ДОПОЛНИТЕЛЬНОМ коде.

Проиллюстрируем принятый для IBM PC (и НЕ только!) формат представления целых чисел. Представим себе, что наша гипотетическая ячейка имеет длину всего 4 бита (один полубайт). Какие ЦЕЛЫЕ числа мы сможем в ней хранить, если будем исходить из принятого формата представления целых чисел?

Из табл. 2.1 видно, что беззнаковые целые (**UnSigned**) представимы только в диапазоне [0,15], а знаковые целые (**Signed**) — в диапазоне [-8,7]. И еще одно — обратите внимание: **десятичные числа из диапазона [8,15] по своему внутреннему (машинному) формату СОВПАДАЮТ с числами в диапазоне [-8,-1].** Об этом же мы вели речь и в примере 2.3.

Таблица 2.1. Значения всех возможных целых чисел формата IBM PC в ячейке длиной 4 бита.

БЕЗзнаковые (UnSigned)			Знаковые (Signed)		
Dec	Bin	Hex	Hex	Bin	Dec
15	1111	F	Эти числа в указанном формате представить НЕВОЗМОЖНО (все биты уже расписаны)		15
14	1110	E			14
13	1101	D			13
12	1100	C			12
11	1011	B			11
10	1010	A			10
9	1001	9			9
8	1000	8			8
7	0111	7	7	0111	7
6	0110	6	6	0110	6
5	0101	5	5	0101	5
4	0100	4	4	0100	4
3	0011	3	3	0011	3
2	0010	2	2	0010	2
1	0001	1	1	0001	1
0	0000	0	0	0000	0
-1	Эти числа в указанном формате представить НЕВОЗМОЖНО (все биты уже расписаны)		F	1111	-1
-2			E	1110	-2
-3			D	1101	-3
-4			C	1100	-4
-5			B	1011	-5
-6			A	1010	-6
-7			9	1001	-7
-8			8	1000	-8

? ВОПРОС.

Что будет, если мы попытаемся втиснуть в эту 4-битовую ячейку число 20 или -12?

? ВОПРОС:

Что получится, если мы напишем, например, на **Borland Pascal**'е такой код:

```
Var   x : Integer;
Begin
   X := 20000 + 20000;
   Writeln ("x = ", x);
End.
```

Какой мы получим результат? Почему? Изменится ли что-то, если мы перейдем на Delphi или Visual C++? Вы легко получите ответ, если поняли примеры 2.1 и 2.2.

2.3. Вещественные числа

Вещественные базовые величины могут быть типа **float (single), double** или **long double (extended)** — см. табл.П4.1-П4.4. Как видно, от платформы они НЕ зависят. Эти вещественные числа обрабатывает *сопроцессор.* Внутреннее (машинное) представление этих чисел достаточно сложное:

S	Характеристика	Нормализованная мантисса

РИС. 2.3. *Формат вещественного числа*

Бит S - как обычно, знак числа.

Характеристика = Смещение ± Порядок

Смещение — число, равное ПОЛОВИНЕ максимально возможного, которое может поместиться в поле **Характеристика**. Таким образом, экономятся место и время, т.к. НЕ нужно выделять разряд для знака порядка и делать дополнительный код для отрицательных порядков.

Размер пространства, которое ПК использует для хранения **Характеристики** и **Мантиссы,** установлен **стандартом IEEE 724-1985 standard for floating point numbers** и поддерживается *всеми* современными компьютерными архитектурами.

Для получения внутреннего представления вещественного числа нужно в любом случае перевести его в двоичный формат.

2.3.1. Представление вещественных чисел в двоичном нормализованном виде

Получив двоичное представление, нужно его *нормализовать*, т.е. двоичное число должно **всегда начинаться с единицы** и иметь следующий вид:

$\pm 1.m_2 * 2^p$,

где m_2 – двоичная мантисса числа.

p – порядок двоичного числа.

 **ЗАМЕЧАНИЕ.**

Если возникнут трудности с получением двоичного порядка, вспомните, как это делается в десятичной системе. Ведь обе системы счисления — позиционные. Аналогия — полная!



ПРИМЕР 2.4.

± 1.0d → ± 1.0b * 2^0

 **ЗАМЕЧАНИЕ.**

Единица — она и в двоичной системе единица!!! Но в вещественном формате она, конечно, отличается от целочисленного формата.

Воспользуемся для дальнейшего изложения результатами примеров 1.3а-1.6а.



ПРИМЕР 2.5.

± 0.5d → ± 0.1b (см. пример 1.3а), в нормализованном виде: **± 1.0b * 2^{-1}**

ПРИМЕР 2.6.

± 0.703125d → ± 0.101101b (см. пример 1.4а) , в нормализованном виде: **± 1.01101b * 2^{-1}**

ПРИМЕР 2.7.

± 0.05d → ± 0.000011(0011)b (см. пример 1.5а) , в нормализованном виде: **± 1.1(0011)b * 2^{-5}**

ПРИМЕР 2.8.

± 117.25d → ± 1110101.01b (см. пример 1.6а) , в нормализованном виде: **± 1.11010101b * 2^6**

Видите ли вы, что десятичная точка "плавает", чтобы обеспечить нормализацию? То идет влево и порядок получает знак плюс, то — вправо, тогда у порядка знак минус. Отсюда и термин — плавающая точка (**floating point**).

2.3.2. Машинные форматы вещественных чисел

Как мы уже выяснили, вещественные базовые величины могут быть типа **float (single)**, **double** или **long double (extended)** — см. табл.П4.1-П4.4. Все они имеют для хранения ячейки *разной* длины. Отсюда и разный диапазон представления для разных типов вещественных чисел, хотя идея хранения почти одна и та же — см. рис. 2.3.

2.3.2.1. Формат float (single)

Число это хранится в ячейке длиной 32 бита, которые распределяются следующим образом:

1– скрытый (невидимый человеку) разряд
V

S	Характеристика			Нормализованная мантисса				
31	30	...	23	22	21	...	1	0
Биты								

РИС. 2.4. *Формат вещественного 32-битного числа.*

На **Характеристику** выделяется **8** битов (разряды 23-30). Максимально возможное шестнадцатеричное число, которое можно разместить в этих битах, равно **FFh**, а его *половина* по определению — это **Смещение = 7Fh.** Поскольку на представление мантиссы остается всего 23 бита (разряды 0-22), а мы знаем, что мантисса всегда нормализованная, возникает вполне резонный вопрос: зачем хранить самую первую единицу мантиссы? Пусть компьютер сам ее восстанавливает... Таким образом, эта самая первая единица в информационные разряды мантиссы НЕ попадает (*скрытый разряд*), а на мантиссу в результате отводится не 23 бита, а 24. Такое представление позволяет верно отображать **7-8** десятичных цифр, а его диапазон представления составляет **1.5 * 10^ – 45 .. 3.4 * 10^38** (Help для Delphi 5). Если посмотреть на табл.П4.1-П4.4, там будет немного другой результат. Разобравшись с приведенными ниже примерами, вы поймете, что точно вычислить допустимый диапазон для вещественных чисел ОЧЕНЬ непростая задача (особенно для ОЧЕНЬ малых чисел, с исчезающим порядком). Даже компиляторы ошибаются...

ПРИМЕР 2.4А.

± 1.0d → ± 1.0 * 2^0 b

Характеристика = 7F + 0 = 7F.

Расписываем подробно содержимое 32-х битов, а затем переводим двоичное представление в шестнадцатеричное.

Положительное число **1.0**:

1 – скрытый разряд
V

0	0	1	1	1	1	1	1	1	0	0	0	0	...	0	0
S	Характеристика								Нормализованная мантисса						
31	30	29	28	27	26	25	24	23	22	21	22	23		1	0
Биты															
3				F				8				0	0 0 0 0		
HEX-код															

РИС. 2.5. *Формат вещественного 32-битного числа 1.0 в Bin и Hex-кодах.*



ОБРАТИТЕ ВНИМАНИЕ!

Скрытая единица НЕ видна человеку, но компьютер о ней знает.

Отрицательное число **-1.0** будет точно такое же, только в знаковом бите **S=1**:

1– скрытый разряд
V

1	0	1	1	1	1	1	1	1	0	0	0	0	...	0	0
S	Характеристика								Нормализованная мантисса						
31	30	29	28	27	26	25	24	23	22	21	22	23		1	0
B				F				8				0	0 0 0 0		

РИС. 2.6. *Формат вещественного 32-битного числа -1.0 в Bin и Hex-кодах.*



ПРИМЕР 2.5А.

$\pm 0.5d \rightarrow \pm 1.0 * 2^{-1} b$

Характеристика = 7F – 1 = 7E.

Опять расписываем содержимое 32-х битов, а затем переводим двоичное представление в шестнадцатеричное. В этом примере изменился только порядок. В мантиссе опять только один разряд, да и тот НЕВИДИМ для человека.

Положительное число **0.5**:

1 – скрытый разряд
V

0	0	1	1	1	1	1	1	0	0	0	0	0	...	0	0
S	Характеристика								Нормализованная мантисса						
31	30	29	28	27	26	25	24	23	22	21	22	23		1	0
3				F				0				0	0 0 0 0		

РИС. 2.7. *Формат вещественного 32-битного числа 0.5 в Bin и Hex-кодах.*

Отрицательное число **-0.5** будет точно такое же, только в знаковом бите **S=1**:

1– скрытый разряд
V

1	0	1	1	1	1	1	1	0	0	0	0	0	...	0	0
S	Характеристика								Нормализованная мантисса						
31	30	29	28	27	26	25	24	23	22	21	22	23	...	1	0
B				F				0				0	0 0 0 0		

РИС. 2.8. *Формат вещественного 32-битного числа -0.5 в Bin и Hex-кодах.*



ПРИМЕР 2.6А.

$\pm 0.703125d \rightarrow \pm 1.01101 * 2^{-1} b$

В этом примере уже более солидная мантисса, а порядок такой же, как и в предыдущем примере. Распишем положительное число, а отрицательное можете сделать сами.

1– скрытый разряд
V

0	0	1	1	1	1	1	1	0	0	1	1	0	1	0	0	0	...	0
S	Характеристика								Нормализованная мантисса									
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	...	0
3				F				3				4				0 0 0 0		

РИС. 2.9. *Формат вещественного 32-битного числа 0.703125 в Bin и Hex-кодах.*

ПРИМЕР 2.7А.

± 0.05d → ± 1.1(0011) * 2^{-5} b

В этом числе совсем другие и порядок, и мантисса.

Характеристика = 7F – 5 = 7A. На этот раз распишем число отрицательное, а с положительным вы вполне способны справиться сами.

1 0111 1010 1 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 ...

^

1

Для удобства перевода двоичного числа в шестнадцатеричную систему счисления разобьем его на тетрады, как мы уже неоднократно делали, и возьмем 8 полубайтов (32 бита):

1011 1101 0100 1100 1100 1100 1100 1100 1100 ...

B D 4 C C C C C

ПРИМЕР 2.8А.

± 117.25d → ± 1.11010101 * 2^{6} b

Характеристика = 7F + 6 = 85. Распишем, например, число положительное:

0 1000 0101 **11010101**00000000000000

^

1

Теперь распишем по тетрадам:

0100 0010 **1110 1010** 1000 0000 0000 0000

4 2 E A 8 0 0 0

Внутреннее представление вещественных чисел, дело, как видите, достаточно кропотливое.

2.3.2.2. Формат double

1– скрытый разряд
V

S	Характеристика			Нормализованная мантисса				
63	62	...	52	51	50	...	1	0
Биты								

РИС. 2.10. *Формат вещественного 64-битного числа.*

Здесь идея та же самая, что и в предыдущем формате представления вещественных данных. На все число отводится ячейка длиной 64 бита. На характеристику расходуется **11** битов (разряды 52-62). Максимально возможное число, которое можно разместить в 11-ти битах: **111 1111 1111b**. Отсюда следует, что

Смещение = 011 1111 1111b → 3FFh.

Мантисса тоже имеет *скрытый* разряд. Такое представление позволяет получить следующий допустимый диапазон для данных типа **double**:

5.0 x 10^—324 .. 1.7 x 10^308

(Help для Delphi-5) и, соответственно, представить **15-16** десятичных цифр.

ПРИМЕР 2.7B.

0.05d → 1.1(0011) * 2^{-5} b

Характеристика = 3FF — 5 = 3FA. Распишем наше число сначала в двоичном виде:

0 011 1111 1010 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001...1001

^

1

Теперь можно получить его и в шестнадцатеричном виде: **3FA999999999999h.** Как видите, информация действительно сжимается...

Потренируйтесь с остальными примерами — это очень полезно для получения навыка в переводе чисел из одной системы счисления в другую.

2.3.2.3. Формат long double (extended)

S	Характеристика			Нормализованная мантисса				
79	78	...	64	63	62	...	1	0
Биты								

РИС. 2.11. *Формат вещественного 80-битного числа.*

На все число отводится 80 битов. Этот формат, как мы узнаем при изучении команд сопроцессора, является основным рабочим форматом для данных сопроцессора. Поэтому, чтобы НЕ делать дополнительных преобразований при вычислениях, **мантисса НЕ имеет скрытого разряда**. На характеристику расходуется **15** битов (разряды 64-78). Максимально возможное число, которое можно разместить в 15-ти битах: **111 1111 1111 1111b**. Отсюда следует, что

Смещение = 011 1111 1111 1111b → 3FFFh.

Такое представление позволяет получить следующий допустимый диапазон для данных этого типа: **3.6 x 10^-4951 .. 1.1 x 10^4932** (Help для Delphi-5) и, соответственно, представить **19-20** десятичных цифр.



ПРИМЕР 2.8С.

± 117.25d → ± 1.11010101 * 2^6 b

Характеристика = 3FFF + 6 = 4005 (при сложении ПОЛНАЯ аналогия с десятичной системой счисления). Расписываем наше число сначала в двоичном коде, а затем в шестнадцатеричном. В результате получаем:

0 100 0000 0000 0101 **1110 1010** 1000 0000 ... .0000b

117.25d → 4005EA8000000000000h

1 100 0000 0000 0101 **1110 1010** 1000 0000 ... .0000b

-117.25d → C005EA8000000000000h

2.4. Простейшая программа описания данных на языке Ассемблера IBM PC для MS DOS

Структура программы на языке Ассемблера зависит от того, для какой платформы (операционной системы) мы пишем программу. Для начала разберемся с простейшим случаем: операционная система MS DOS, программа для резервирования памяти и описания данных. Предварительно познакомимся **на практике** и с некоторыми ***новыми понятиями***: *сегмент* (более подробно с сегментами мы познакомимся в п. 3.3.2), *модель памяти* (см. п. 3.4.1) применительно к компилятору **Borland TASM-5.0.**

2.4.1. Формат директив и машинных команд

Язык Ассемблера состоит из *директив* и *машинных команд*. Они отдают приказы компьютеру, что ему делать. С ними мы тоже будем знакомиться постепенно, по мере необходимости. Структура директив и команд одна и та же и состоит из следующих полей:

Между полями (кроме комментария) нужен хотя бы ОДИН пробел. Обычно стараются поля выравнивать в столбик (используя для этого клавишу TAB), что повышает читабельность программы.

Имя	Мнемокод	Операнды	; Комментарии

РИС. 2.12. *Формат директив и машинных команд.*

Поскольку язык Ассемблера был призван упростить и очеловечить написание программ в кодах компьютера, все директивы и команды имеют мнемонический код (*мнемокод*), описывающий сокращенно (или полностью) назначение директивы или команды на национальном языке (в нашем случае — на английском). А были времена, когда мнемокоды были на русском языке... И Ассемблер назывался Автокод.

Наличие мнемокода является ОБЯЗАТЕЛЬНЫМ элементом формата, все остальные — НЕ обязательны и зависят от самой команды или директивы. Директивы обычно что-то в программе описывают (данные, режимы и проч.). А мнемокод команды — это сама машинная команда в ее человеческом варианте. Поскольку **Turbo Assembler (TASM)** в действительности содержит в себе два синтаксических стандарта языка Ассемблера — **MASM** и **Ideal,** некоторые директивы могут быть написаны по-разному. Эквиваленты директив в этих двух режимах приведены в Приложении 5. Остальные директивы выглядят одинаково как для режима **MASM,** так и **Ideal**.

Комментарии могут быть и самостоятельными строками программы на Ассемблере. Комментариям в любом языке программирования уделяется серьезное внимание, а в языке Ассемблера и подавно! В нем так много мелочей, которые легко подзабыть...

2.4.2. Директивы инициализации и описания данных на языке Ассемблера

Данные могут размещаться в участках памяти, которые называются *сегменты*. Обычно это или *сегмент данных*, или *сегмент кода*. Сегменты описываются с помощью всепогодной директивы Ассемблера **SEGMENT** или с помощью современных, упрощенных директив **.Model**, **.Code** или **.Data**.

Для инициализации **простых** типов данных в Ассемблере используются специальные директивы **Dx**, являющиеся указаниями компилятору на выделение определенных объемов памяти. Для языка Ассемблера имеет значение только *длина ячейки*, в которой размещено данное, а какое это данное — зависит всецело от человека, пишущего программу его обработки.

Таблица 2.2. Директивы Ассемблера для задания простых типов данных.

Длина (бит)	Инициализация	Описание
8	DB	BYTE
16	DW	WORD
32	DD	DWORD
64	DQ	QWORD
80	DT	TBYTE

Мнемокоды директив инициализации данных **Dx** означают следующее:

DB (Define Byte) — определить байт.

DW (Define Word) — определить слово.

DD (Define Double Word) — определить двойное слово.

DQ (Define Quarter Word) — определить учетверенное слово.

DT (Define Ten Bytes) — определить 10 байтов.

Отсюда понятно, что означают мнемокоды директив описания данных.

Для директив инициализации данных **имя** может быть, а может и отсутствовать. Если имя есть, компьютер с ним связывает *адрес памяти*, и в дальнейшем в командах мы можем использовать это имя по своему усмотрению. Например, заносить по указанному именному адресу какое-то значение или извлекать из него хранимое значение.

Регистр букв для имен и директив в Ассемблере безразличен, как и в алгоритмическом языке Pascal, поэтому в наших примерах мы их будем писать по-разному, руководствуясь НАГЛЯДНОСТЬЮ представления. **Но при стыковке программ на Ассемблере с программами на языке С/С++ регистр для имен переменных имеет ОЧЕНЬ большое значение**, о чем в свое время еще будет сказано.

ПРИМЕР 2.9.

Напишем, как выглядят операции описания и инициализации данных в разных языках программирования. В Ассемблере это соответствует распределению и инициализации памяти. При распределении памяти обычно мы знаем, с каким форматом данных собираемся работать, но пока можем НЕ знать, какое значение будет храниться в той или иной ячейке памяти, — в этом случае в поле операнда директивы **Dx** нужно поставить знак вопроса. А если нужно выделить непрерывный участок памяти из нескольких ячеек, пишется параметр коэффициента повторений **dup**.

Pascal	C/C++	Assembler
N: **Integer**;	**int** N; // Win16 // Win32	N dw ? N dd ?
A: **single**;	**float** A;	A dd ?
B: **double**; ... B := -898.6897;	**double** B = -898.6897;	B DQ -898.6897
Arr:**array** [1..100] **of extended**;	**long double** Arr [100];	Arr DT 100dup(?)
Mas:**array** [1..10] **of byte**; .... Mas [1]:=1; Mas [2]:=2; .... Mas [9]:=9; Mas [10]:=10;	**unsigned char** Mas [] = {1,2,3,4,5,6,7,8,9,10};	Mas DB 1,2,3,4 DB 5,6,7,8,9,10
MasW:**array** [1..10] **of word**; .... Mas [1]:=0; Mas [2]:=0; .... Mas [9]:=0; Mas [10]:=0;	**unsigned short int** MasW [] = {0,0,0,0,0,0,0,0,0,0};	MasW DW 10 dup(0)

В данном примере все ячейки памяти получились поименованными, т.е. с ними могут работать машинные команды и директивы. Есть различные способы адресации, с которыми мы познакомимся позднее, а пока усвоим тот факт, что ячейки памяти могут иметь имена, что ОЧЕНЬ упрощает процесс программирования на Ассемблере. Как и в любом языке программирования, желательно, чтобы эти имена отражали суть описываемого Вами в программе действия. Например, имена переменных в каком-то арифметическом выражении.

2.4.3. Общая структура программы на Ассемблере в MS DOS

Заголовок программы	**Title** Заголовок программы
Модель памяти	**.Model** large **.Model** tiny **.Model** small **.Model** medium **.Model** compact **.Model** huge
Сегмент стека	**.STACK**
Директивы распределения и инициализации памяти	**Dx**
Сегмент данных	**.DATA**
Директивы распределения и инициализации памяти для переменных	**Dx**
Директивы описания внешних переменных	**Extrn**
Сегмент кода	**.CODE**
Директивы распределения и инициализации памяти для переменных	**Dx**
Директивы описания внешних ссылок и/или переменных	**Extrn**
Процедуры	**Proc** ... **endP**
Конец программы	**END**

РИС. 2.13. *Общая структура asm-модуля в MS DOS.*

Программа на Ассемблере может начинаться с НЕОБЯЗАТЕЛЬНОГО элемента — заголовка и всегда должна заканчиваться директивой **END**. Наличие сегментов и их количество зависят от специфики решаемой задачи. Оформляется программа в виде модуля с расширением имени **asm**. Будем этот исходный файл называть asm-файл. Чтобы получить машинный код, asm-файл нужно откомпилировать (ассемблировать). Если НЕТ синтаксических ошибок, в результате получим машинный код — файл с расширением obj (объектный файл). Этот файл сам по себе выполняться НЕ может. Пока для наших целей он и НЕ нужен. Если в исходном asm-файле есть ошибки, то obj-файл НЕ выдается.

В дистрибутиве TASM-5 есть несколько ассемблеров. Нам пока нужен 16-разрядный компилятор. Это файл **tasm.exe,** который соответствует версии 4.1. Всегда можно получить файл листинга (lst-файл) и сделать анализ машинного кода или ошибок. Делается это с помощью ключа компиляции **/L (listing):**

tasm.exe имя**.asm/L**

2.4.4. Структура lst-файла в TASM

Turbo Assembler Version 4.1 20/05/01 23:34:06 Page 1 Имя.ASM Title-информация			
1	**2**	**3**	**4**
№ строки программы	**Адрес (смещение)**	**Машинный код**	**Программа на Ассемблере – исходный код**
Turbo Assembler Version 4.1 20/05/01 23:34:06 Page N **Symbol Table** Таблица распределения памяти			
Groups & Segments Bit Size Align Combine Class Информация о сегментах и их длине			

РИС. 2.14. *Общая структура lst-файла.*

Поля 2 и 3 содержат HEX-коды. Поле 3 (машинный код) — это и есть результат ассемблирования (компиляции на Ассемблере). Эти кòды можно увидеть, если просмотреть объектный или исполняемый (com- или exe-файлы) через просмотрщик (клавиша **F3,** например, в оболочках **FAR** или в **Windows Commander**) или пропустить его через отладчик (**Debug**).

2.4.5. Пример программы на языке Ассемблера для проверки внутреннего представления данных

ПРИМЕР 2.10.

Напишем теперь на языке Ассемблера программу с указанием всех данных, с машинным представлением которых мы встретились в главе 2. Назовем этот файл **Asm_digt.asm**. Поскольку данные могут находиться либо в сегменте данных, либо в сегменте кода, определимся, например, на сегменте данных (для данного примера это НЕ принципиально). Модель для этого примера тоже безразлична (что это вообще такое, мы разберем подробно в п. 3.4.2).

Исходный текст программы Asm_digt.asm

```
; Файл Asm_digt.asm
title labl-2 (CopyRight by Голубь Н.Г., 1993-1997)
;------------------------------------------------------------------
; 2.4.5. Пример программы на языке Ассемблера для
;        проверки внутреннего представления данных
;------------------------------------------------------------------
.MODEL   tiny ; эту директиву можно писать и БЕЗ точки
   ;          Модель для этого примера БЕЗРАЗЛИЧНА
; Могут быть еще модели:
;   small medium compact large huge
.DATA    ; писать БЕЗ точки эту директиву НЕЛЬЗЯ!!!
;=============== процессор i8086 ====================
;------ byte ----------
i      db   0,   255
;------ word ----------
iw     dw   40000
        dw  65535
;------- shortint -------
is     db   -128, 127
;------- integer -------
ii     dw   -32768
       dw   -25536
       dw   32767
       dw   -1607
;------ longint ---------
il     dd   4294965689
iil    dd   -1607
;=============== сопроцессор i8087 ================
;------- single ----------
a      dd   -1.
b      dd   1.
ai     dd   -0.5, 0.5
       dd   0.703125
       dd   -0.703125
       dd   0.05
       dd   -0.05
       dd   117.25
       dd   -117.25
;------ double -----------
ad     dq   -1.
bd     dq   1.
aid    dq   -0.5, 0.5
       dq   0.703125
       dq   -0.703125
       dq   0.05
```

```
        dq    -0.05
        dq    117.25
        dq    -117.25
;——     extended ————————
ae      dt    -1.
be      dt    1.
aie     dt    -0.5, 0.5
        dt    0.703125
        dt    -0.703125
        dt    0.05
        dt    -0.05
        dt    117.25
        dt    -117.25
;————     дл.целое ————-
iq      dq    4294965689
iiq     q     -1607
        dq    6858569895795822234 ; а этого данного в главе 2 НЕ было!
              END
```

Теперь эту программу надо откомпилировать и получить листинг:

tasm.exe Asm_digt.asm/L.

В результате получаем файл **Asm_digt.lst**. Рассмотрим нужный нам фрагмент листинга:

Фрагмент файла листинга Asm_digt.lst

```
Turbo Assembler     Version 4.1     20/05/01 22:31:08          Page 1
ASM_DIGT.ASM
lab1-2 (CopyRight by Голубь Н.Г., 1993-1997)
  8   0000              .DATA ; писать БЕЗ точки эту директиву НЕЛЬЗЯ!!!
  9                           ;=============== процессор i8086
 10                           ;————— byte ——————————
 11   0000   00    FF                           i     db  0, 255
 12                           ;————— word ——————————
 13   0002  9C40                                iw    dw  40000
 14   0004  FFFF                                      dw  65535
  5                           ;——————— shortint ————————-
 16   0006  80 7F                               is    db  -128, 127
 17                           ;——————— integer ————————-
 18   0008  8000                                ii    dw  -32768
 19   000A  9C40                                      dw  -25536
 20   000C  7FFF                                      dw  32767
 21   000E  F9B9                                      dw  -1607
 22                           ;————— longint —————————-
 23   0010  FFFFF9B9          il                dd  4294965689
 24   0014  FFFFF9B9                            iil   dd  -1607
 25                           ;=============== сопроцессор i8087
 26                           ;——————— single ———————————
 27   0018  BF800000                            a     dd -1.
 28   001C  3F800000                                  dd  1.
```

```
29  0020 BF000000 3F000000                  ai   dd  -0.5, 0.5
30  0028  3F340000                               dd  0.703125
31  002C  BF340000                               dd  -0.703125
32  0030  3D4CCCCD                               dd  0.05
33  0034  BD4CCCCD                               dd  -0.05
34  0038  42EA8000                               dd  117.25
35  003C  C2EA8000                               dd  -117.25
36                          ;———— double ————————
37  0040 BFF0000000000000                   ad   dq -1.
38  0048 3FF0000000000000                   d    dq  1.
39  0050 BFE0000000000000   +               aid  dq  -0.5, 0.5
40         3FE0000000000000
41  0060 3FE6800000000000                        dq  0.703125
42  0068 BFE6800000000000                        dq  -0.703125
43  0070 3FA999999999999                         dq  0.05
44  0078 BFA999999999999                         dq  -0.05
45  0080 405D500000000000                        dq  117.25
46  0088 C05D500000000000                        dq  -117.25
47                          ;—————————extended ————————
48  0090 BFFF8000000000000000               ae   dt -1.
49  009A 3FFF8000000000000000               be   dt  1.
50  00A4 BFFE8000000000000000+              aie  dt  -0.5, 0.5
51         3FFE8000000000000000
52  00B8 3FFEB400000000000000                    dt  0.703125
53  00C2 BFFEB400000000000000                    dt  -0.703125
54  00CC 3FFACCCCCCCCCCCCCCCD                    dt  0.05
55  00D6 BFFACCCCCCCCCCCCCCCD                    dt  -0.05
56  00E0 4005EA80000000000001                    dt  117.25
57  00EA C005EA80000000000001                    dt  -117.25
58                          ;————— дл.целое ————————
59  00F4 00000000FFFFF9B9                   iq   dq  4294965689
60  00FC FFFFFFFFFFFFF9B9                   iiq  dq  -1607
61  0104 0984A74C569C1B1A dq 685856989579582234; а этого данного в главе 2 НЕ было!
62                          END
```

Проанализируем полученный результат. Строки 23, 24 показывают, что действительно, с точки зрения компьютера числа 4294965689 и -1607 имеют одно и то же внутреннее представление в формате двойного слова: **FFFFF9B9h** (см. пример 2.3). Внутренние представления чисел 0.05 и -0.05 (см. строки 32, 33) в младшем полубайте отличаются от полученных значений в примере 2.7а: было 3D4CCCCCh и BD4CCCCDh, стало **3D4CCCCDh** и **BD4CCCCDh.** Произошло это от ОКРУГЛЕНИЯ (см. внимательно пример 2.7а — машинное **двоичное** представление). То же самое и в строках 54, 55. А вот откуда взялась младшая единица для чисел 117.25 и -117.25 (строки 56, 57) — это загадка компилятора TASM. Видите, как трудно с вещественными числами (даже такими простыми)! И компиляторы ошибаются...

Ради любопытства откомпилируем файл **Asm_digt.asm** с помощью компилятора фирмы Microsoft MASM версии 6.12: **masm.exe Asm_digt.asm/L**.

Листинг, в общем, аналогичен предыдущему, но для чисел 117.25 и -117.25 получился НОРМАЛЬНЫЙ результат (молодцы ребята из компании Microsoft!):

```
00E0  4005EA8000000000000       dt  117.25
00EA  C005EA8000000000000       dt  -117.25
```

Строки в листинге masm-компилятор НЕ нумерует.

3

Архитектура IBM PC

Невежественными бывают только те, которые решаются такими оставаться.

Платон (428 или 427–348 или 347 гг. до н.э.)

Компьютеры сегодня известны каждому. Среди них особенно популярны персональные компьютеры (ПК), например, такие, как наш IBM PC. Первый IBM PC появился осенью 1981 года и быстро стал ОЧЕНЬ популярным. Сейчас ПК продолжают развиваться, наращивая мощность и быстродействие, благодаря применению новейших "железных" технологий. *Неуклонное повышение быстродействия микропроцессоров стало таким же неотвратимым явлением, как смерть и налоги...*, — пишет в одном из своих выпусков популярный компьютерный журнал **PC Magazine/Russian Edition** (http://www.pcmag.ru/news.asp?ID=590). Чтобы понять работу машинных команд, надо хотя бы немного разбираться в том, что же такое компьютер и как он работает. А чтобы писать эффективные программы, обслуживающие нижний уровень компьютера, надо архитектуру ПК знать в совершенстве. Ну, хотя бы так, как Питер Нортон [Л2-4, Л4-9], тогда и у Вас будут свои утилиты и кое-что еще...

3.1. Семейство IBM PC

Понимание современных процессоров ПК требует некоторых познаний из истории развития их семейства. Причина этой необходимости заключается в подобии и совместимости, благодаря которым множество различных процессоров объединяются в семейство. Компания Intel назвала серию своих микросхем центрального процессора для ПК семейством x86. Мы до сих пор привязаны к конструктивным решениям, найденным разработчиками Intel около четверти века назад. При этом наиболее значительным является метод, принятый разработчиками Intel для расширения адресного пространства, который не идет ни в какое сравнение с используемым в более ранних моделях микропроцессоров, — Питер Нортон [Л2-4].

Перечислим **основные вехи** эволюции семейства x86, поскольку сейчас буквально каждый день приносит все более и более впечатляющие известия о появлении новых, более мощных и производительных микропроцессоров клона x86 от разных производителей. На момент написания данной книги (май 2001 г.) основная борьба в этом плане была между корпорацией Intel и AMD (Advanced Micro Devices). В печати можно было видеть такие статьи:

- **ВЫПУСК 64-РАЗРЯДНОГО AMD HAMMER ЗАДЕРЖИВАЕТСЯ!** Первый процессор AMD нового поколения ClawHammer выйдет не раньше конца будущего года (2002), но Intel не до смеха, так как с Pentium 4 тоже не все гладко. (http://zdnet.ru/news.asp?ID=2469)
- **INTEL ОТВЕЧАЕТ AMD ПРОЦЕССОРОМ TUALATIN.** В июле этого года (2001) в ответ на новый Athlon 4 компании Advanced Micro Devices микропроцессорный гигант выпустит пять новых мобильных Pentium III. (http://zdnet.ru/news.asp?ID=2502)
- **PENTIUM 4: ТЕСТ НА СООБРАЗИТЕЛЬНОСТЬ?** Зачем покупать процессоры Intel Pentium 4 1,3 ГГц, если Pentium III 1 ГГц намного быстрее и дешевле? (http://zdnet.ru/news.asp?ID=2115)

Компания Intel больше не имеет полного контроля над развитием семейства x86. Она все еще выпускает большую часть микросхем ЦП семейства x86, но теперь любой изготовитель может предлагать новые разработки, и, если они достаточно хорошо себя покажут, остальные производители их переймут, а разработчики программ создадут соответствующее обеспечение, — Питер Нортон [Л2-4]. Такими разработчиками микросхем на сегодняшний день являются фирмы: AMD (30% рынка), Cyrix, IDT и Rise.

Из табл. 3.1. видно, чем отличалась каждая новая модель от предыдущей.

Таблица 3.1. Эволюция семейства x86.

Тип микропроцессора	Год появления	Некоторые характеристики			Основные особенности
		Разрядность		Встроенный сопроцессор (FPU)	
		Данные	Адрес		
8086	1978/1979	16/16	20	Нет	Реальный режим работы процессора. Верхний предел доступной памяти 1Мб.
8088 (IBM PC/XT)	1979/1980	8/16	20	Нет	Более дешевый процессор
286 (IBM PC/AT)	1982	16/16	24	Нет	Элементы защищенного режима. Верхний предел доступной памяти 16Мб. Добавлены команды.

Таблица 3.1. (Продолжение)

386DX	1985	32/32	32	Нет	32-разрядное программирование. Защищенный и виртуальный (V86) режимы работы процессора. Страничная организация памяти (до 4Гб)
386SX	1988	32/16	32	Нет	Более дешевый процессор
486DX	1989	32/32	32	Да	В одном корпусе ЦП, FPU и кэш-память первого уровня (L1). Конвейерная архитектура. Ориентация на сети.
486SX	1991	32/32	32	Нет	Более дешевый процессор
486DX2	1992	32/32	32	Да	Производительность больше на 70%, чем у 486DX
Pentium	1993	64/64	32	Да	Первый **именованный** процессор семейства. Удвоен объем L1. Конвейерная обработка команд. Суперскалярное повышение производительности.
486DX4	1994	32/32	32	Да	Улучшение производительности по сравнению с 486DX2

Таблица 3.1. (Продолжение)

Тип микропроцессора	Год появления	Некоторые характеристики			Основные особенности
		Разрядность		Встроенный сопроцессор (FPU)	
		Данные	Адрес		
Pentium Pro (P6)	1995	64/64	32/36	Да	Микроархитектура P6. Кэш-память второго уровня (L2). Динамическое выполнение команд.
Pentium MMX	1996	64/64	32	Да	Новые команды MMX. Увеличен объем L1.
Pentium II	1996	64/64	32/36	Да	Новое конструктивное исполнение Pentium MMX и подключение к системной плате. Расширение адресного пространства до 64Гб.
Pentium II Xeon (Высший класс - High-End)	1997	64/64	32/36	Да	Началось деление рынка ПК на серверы, рабочие станции, настольные и мобильные. Увеличен объем L2, она работает с полной тактовой частотой процессора. Усовершенствована поддержка мультипроцессорных систем.
Pentium II Celeron (Low-End)	1998	64/64	32/36	Да	Более дешевая версия Pentium II Xeon за счет изменения конструктивного исполнения (уменьшены размер кэша L2 и скорость шины).
AMD Athlon (K6-2)	1998	64/64	32/36	Да	Новые команды расширения MMX: 3D.

Таблица 3.1. (Продолжение)

Тип микропроцессора	Год появления	Некоторые характеристики			Основные особенности
		Разрядность		Встроенный сопроцессор (FPU)	
		Данные	Адрес		
AMD Duron (Low-End)	1998	64/64	32/36	Да	Более дешевая версия AMD Athlon за счет изменения конструктивного исполнения.
Pentium III Katmai, Xeon (High-End)	1999	64/64	32/64	Да	Команды SSE (Streaming SIMD Extension)- расширение технологии MMX.
AMD Athlon (Thunderbird, K6-3)	1999	64/64	32/64	Да	Новые команды расширения MMX: Enhanced 3DNow! Более производителен, чем Pentium-III. Постоянно идут конструктивные изменения в сторону увеличения производительности
AMD Duron (K6-3, Low-End)	1999	64/64	32/36	Да	Более дешевая версия AMD Athlon (K6-3) за счет изменения конструктивного исполнения.
Pentium III Willamette	2000	64/64	32/64	Да	Команды SSE-2 (для видео и синтеза речи). Новый набор инструкций SIMD2. Механизм неупорядоченного суперскалярного выполнения инструкций P6 (Advanced Dynamic Execution), ядро с двойным возбуждением, кэш трассировки, . усовершенствованн ые арифметические устройства для операций

Таблица 3.1. (Продолжение)

Тип микропроцессора	Год появления	Некоторые характеристики			Основные особенности
		Разрядность		Встроенный сопроцессор (FPU)	
		Данные	Адрес		
Pentium III Willamette	2000	64/64	32/64	Да	с плавающей запятой и обработки мультимедиа, 400-МГц шина данных, встроенный кэш второго уровня и 20-ступенчатый конвейер.
Pentium IV	15 ноября 2000 (Москва). 20 ноября 2000 (корпорация Intel)	64/64	64	Да	Принципиально новая микроархитектура Intel NetBurst – развитие Pentium III Willamette.
AMD Athlon 4 (Palomino)	Май 2001 мобильная версия	64/64	64	Да	Потребляет меньше энергии, чем Athlon, и поэтому перспективен для мобильных ПК стандартного размера. Процессор одинаков для мобильных ПК, рабочих станций и серверов. Двухпроцессорная архитектура. Команды 3DNow! Professional, обеспечивающие совместимость с SSE.
AMD Duron 4 (Morgan) (Low-End)	Май 2001 мобильная версия	64/64	64	Да	Более дешевая версия AMD Athlon 4 за счет изменения конструктивного исполнения.
Pentium IV Xeon (High-End)	Май 2001	64/64	64	Да	Первое поколение процессоров Xeon с архитектурой NetBurst

В некоторых источниках даны немного другие даты появления микропроцессоров. Это может быть связано с годом реальной эксплуатации данного микропроцессора в составе ПК. **Каждая новая модель поддерживает предыдущую на уровне машинных команд, чем достигается подобие и совместимость при выполнении программ**. Все новые термины можно посмотреть в Глоссарии.

Почитаем, что было написано по поводу Pentium IV в статье Андрея Борзенко "ДОЛГОЖДАННАЯ ПРЕМЬЕРА PENTIUM 4", http://www.pcweek.ru/news.asp?ID=4784.

20 ноября корпорация Intel официально представила микропроцессор для высокопроизводительных настольных ПК Pentium 4.

Он изготовлен на основе микроархитектуры NetBurst, обеспечивающей новый уровень производительности трехмерной графики, Интернет-приложений, а также средств кодирования аудио- и видеоданных. Микросхема, содержащая 42 млн. транзисторов, занимает площадь 217 кв. мм. Кристаллы в 423-контактных корпусах PPGA с тактовыми частотами 1,4 и 1,5 ГГц производятся с соблюдением проектных норм 0,18 мкм.

Ключевыми особенностями NetBurst являются гиперконвейерная микроархитектура Hyper Pipelined и технология ядра быстрого исполнения Rapid Execution Engine. Удвоенная до 20 ступеней длина конвейера (по сравнению с Pentium III) существенно повышает производительность нового микропроцессора и увеличивает резерв по тактовой частоте.

Арифметико-логические устройства нового микропроцессора работают на удвоенной по сравнению с ядром тактовой частоте. Это позволяет выполнять некоторые инструкции за половину такта, что повышает скорость обработки и сокращает задержки при вычислениях.

Системная шина с частотой 400 МГц обеспечивает втрое большую пропускную способность, чем шина процессора Pentium III, так что скорость передачи данных между Pentium 4 и контроллером памяти достигает 3,2 Гб/с. В кэш-памяти (первого уровня) линии исполнения (Execution Trace Cache) хранятся дешифрованные инструкции, благодаря чему в программных циклах выполнения команд устраняются задержки, связанные с их декодированием.

Кроме того, новая архитектура предусматривает кэш-память второго уровня типа Advanced Transfer Cache объемом 256 Кб. Кристалл Pentium 4 включает усовершенствованный механизм динамического исполнения (Advanced Dynamic Execution), который обеспечивает оптимальную загрузку вычислительных устройств.

Микропроцессор содержит улучшенные схемы предсказания ветвлений, позволяющие ему выполнять программы в нужной последовательности и снижающие потери времени, связанные с ошибочным выбором последовательности вычислений. Потоковые SIMD-расширения 2 (SSE2) добавляют к технологиям MMX и SSE 144 новые команды, в том числе ***128-разрядные целочисленные инструкции и 128-разрядные инструкции для вычислений с плавающей точкой.***

Основой для платформ на базе Pentium 4 стал набор микросхем i850, поддерживающий память типа RDRAM (Rambus Direct RAM) PC600 и PC800. В качестве «южного» моста в нем используется микросхема ICH2. Кристалл «северного» моста (MCH) требует охлаждающего радиатора. Набор i850 поддерживает два канала Rambus. В зависимости от режима передачи и тактовой частоты пропускная способность системной шины и каналов памяти составляет 3,2 Гб/с. Поддерживаемый объем памяти с контролем и исправлением ошибок (ECC) может достигать 2 Гб.

Тест SPEC CPU 2000 (SPECint2000 — 544, SPECfp2000 — 562) показал, что Pentium 4 обладает наивысшей производительностью среди всех процессоров Intel для настольных ПК. Стоимость новых микросхем (при поставках партиями по тысяче штук) составляет соответственно 819 и 644 долл. для моделей с тактовыми частотами 1,5 и 1,4 ГГц.

Итак, в Pentium 4 появляются новые команды, частично опробованные в Pentium III Willamette, растет разрядность представления целочисленных и вещественных данных, удваивается длина конвейера, что позволяет быстрее обрабатывать поступающие в микропроцессор команды. И, конечно, это еще не предел. Продолжение следует...

3.2. Основные блоки IBM PC

Принципиально все современные компьютеры базируются на *архитектуре фон Неймана* — совместное хранение данных и программ. Она включает в себя три основные составляющие: **центральный процессор** (ЦП, CPU — Central Processing Unit), **оперативное запоминающее устройство** (ОЗУ), **внешние запоминающие устройства** (ВЗУ). Конечно, мозгом всей системы является ЦП и его базовая архитектура. Именно ЦП обрабатывает все команды. У семейства IBM PC ЦП и ОЗУ размещаются на материнской плате (раньше для первых моделей она называлась системной платой). Материнская плата является платформой для построения ПК, от ее возможностей зависят и возможности ПК. На этой плате есть соответствующие разъемы, к которым может быть подсоединено огромное количество самых разнообразных ВЗУ: дисплей, клавиатура, принтер, дисководы, CD-ROM, DVD-ROM, джойстик, модем и проч.

Еще в конце 40-х годов фон Нейман сказал: *Память определяет быстродействие*. В настоящее время существуют шесть **основных** видов памяти: оперативная (ОЗУ, RAM — Random Access Memory), регистры (Registers), постоянная (ROM — Read-only Memory), флэш-память (Flash RAM), процессорная или дисковая кэш-память (Cache Memory) и внешняя (ВЗУ). Различаются они своим назначением (см. Глоссарий) и быстродействием. Процессорная сверхбыстрая кэш-память (L1 и L2) и регистры расположены внутри процессора, поэтому они отличаются высоким быстродействием (область наносекунд — миллионных долей секунды). RAM обычно работает в области микросекунд, но сейчас технология производства настолько усовершенствовалась, что отдельные виды RAM по скорости сравнимы с регистрами. Почитайте, например, статью "КОГДА ПАМЯТЬ ОБГОНЯЕТ ПРОЦЕССОР" — http://www.pcweek.ru/news.asp?ID=5816 (март 2001 г.). Внешняя память по-прежнему является низкоскоростной (область миллисекунд — тысячных долей секунды), хотя и тут прогресс не стоит на месте.

Команды и данные хранятся либо в ОЗУ, либо в ВЗУ. В любом случае, когда программа выполняется, она должна находиться в ОЗУ. Обмен информацией между процессором и ОЗУ происходит через адресную шину и/или через шину данных в зависимости от вида обрабатываемой информации. От пропускной способности этих шин зависит фактическая скорость Ваших вычислений и размер адресуемой памяти — подробнее см. п. 3.4.

3.3. Регистры IBM PC XT

Регистры очень важны для любого процессора. **Адресов у них нет, но зато есть имена**. Они считаются быстрой памятью, поскольку находятся в составе самого процессора. Правда, в мире IBM PC все относительно...

Первые модели 8086/8088 имели всего 14 16-разрядных регистров. А Pentium II, например, содержит уже гораздо большее число регистров, большинство из них 64-разрядные. Кроме того, есть регистры, обеспечивающие конвейерную обработку машинных команд, которые программам НЕ видны, а видны только самому хозяину — процессору.

В архитектуру IBM PC в самом начале были заложены возможности для развития. И все следующие семейства процессоров их расширяли (**extension** — расширение). Вначале появились возможности для 32-разрядного программирования (i386) — это расширение 16-разрядного программирования, затем расширение команд сопроцессора в виде дополнительного набора команд MMX (Multimedia Extensions), далее опять расширение, на этот раз набора команд MMX — команды SSE (Streaming SIMD Extension) и так далее, до бесконечности. Современные расширения Pentium 4, как мы уже видели, касаются в основном мультимедиа. Эта технология широко применяется сейчас в компьютерной графике, кино (вспомним "Матрицу" или "Прогулки с динозаврами!"), телевидении, образовании, а также активно осваивается операционными системами Windows, которые тоже непрерывно расширяются...

Чтобы все это понимать, сначала разберемся с 14 базовыми 16-разрядными регистрами. Регистры в старших семействах их расширяют. В некоторых случаях добавляются свои, специфические регистры, реализующие машинные команды или режимы, которые в младших моделях отсутствовали.

3.3.1. Регистры общего назначения

Регистры общего назначения (РОН) являются основными рабочими регистрами ассемблерных программ — "рабочими лошадками". Их отличает то, что к ним можно адресоваться одним словом (16 битов) или однобайтовым кодом (8 битов). Левый байт считается старшим (**High**), а правый — младшим (**Low**).

High Биты Low															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Регистр AX (Accumulator)															
AH								**AL**							
Регистр BX (Base Register)															
BH								**BL**							
Регистр CX (Count Register)															
CH								**CL**							
Регистр DX (Data Register)															
DH								**DL**							

РИС. 3.1. *Регистры общего назначения процессора i8086/i8088.*

Регистр AX = <AH:AL> — первичный аккумулятор, используется во всех операциях ввода/вывода, в некоторых операциях со строками и в некоторых арифметических операциях. Например, команды умножения и деления предполагают ОБЯЗАТЕЛЬНОЕ использование регистра AX (AL). Некоторые команды генерируют более эффективный код, если они имеют ссылки на регистр AX (см таблицу п. 7.1).

Регистр BX = <BH:BL> — базовый регистр, единственный из регистров общего назначения, используемый в индексной адресации. Кроме того, регистр BX может использоваться при вычислениях.

Регистр CX = <CH:CL> является счетчиком. Он необходим для управления числом повторений циклов и для операций сдвига влево или вправо. Регистр CX может использоваться также для вычислений.

Регистр DX = <DH:DL> — регистр данных. Используется в некоторых операциях ввода-вывода, в операциях умножения и деления 16-разрядных чисел совместно с регистром AX.

Любой из регистров общего назначения может быть использован для суммирования или вычитания 8- или 16-разрядных величин.

3.3.2. Сегментные регистры

Базовых сегментных регистров — четыре. Зачем они нужны и что это вообще такое, мы будем выяснять постепенно, а пока дадим их определения. Заметим только, что ВСЕ сегментные регистры служат для указания *начала* соответствующего сегмента.

Регистр CS (Code Segment) — регистр сегмента кода, содержит начальный адрес сегмента кода. Этот адрес плюс величина смещения в командном указателе (регистр IP — см. п. 3.3.3) определяют адрес команды, которая должна быть выбрана для выполнения. Для обычных программ нет необходимости делать ссылки на регистр CS.

Регистр DS (Data Segment) — регистр сегмента данных, содержит его начальный адрес. Этот адрес плюс величина смещения, определенная в команде, указывают на конкретную ячейку в сегменте данных.

Регистр SS (Stack Segment) — регистр сегмента стека, содержит начальный адрес в сегменте стека.

Регистр ES (Extra Segment) — регистр сегмента расширения. Некоторые операции над строками используют дополнительный сегментный регистр ES для управления адресацией памяти. В этом случае регистр ES связан с индексным регистром DI. Обычно эту связь обозначают парой: <ES:DI>. Если необходимо использовать регистр ES, ассемблерная программа должна его инициализировать явно.

3.3.3. Регистр указателя команд

Регистр IP (Instruction Pointer) содержит смещение на команду, которая должна быть выполнена. Иначе говоря, данный регистр совместно с регистром CS (<CS:IP>) содержит **адрес СЛЕДУЮЩЕЙ команды**. Обычно он в программе явно не используется, но может изменять свое значение при использовании отладчика DOS DEBUG для тестирования программы. Именно благодаря содержимому этого регистра процессор всегда "знает, что ему делать дальше".

3.3.4. Регистр флагов

Это ОЧЕНЬ важный регистр, который сигнализирует процессору о его состоянии или о том, как выполнилась та или иная арифметическая или логическая команда. Этот регистр своего имени НЕ имеет, но в различных отладчиках он обычно называется FLAGS. Девять из 16 битов регистра флагов являются активными. Флаг — это бит, принимающий значение **1**, если он ***установлен***, и **0**, если он ***сброшен***. За битами регистра флагов закреплены соответствующие имена, облегчающие понимание их назначения (для знающих английский язык).

Регистр флагов (Flags)															
Биты															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

РИС. 3.2. *Регистр флагов процессора i8086/i8088.*

Таблица 3.2. Назначение битов регистра Flags.

Бит	Назначение
Флаги условий	
CF (Carry Flag)	Флаг переноса. Содержит признак переноса информации из старшего бита (после арифметических операций), а также последний бит при операциях сдвига.
PF (Parity Flag)	Флаг четности. Он устанавливается в 1, если в восьми младших битах результата очередной команды содержится четное число единиц. Это учитывается в операциях ввода-вывода.
AF (Auxiliary Carry Flag)	Флаг дополнительного переноса. Фиксирует особенности работы операций над двоично-десятичными данными (BCD).
ZF (Zero Flag)	Флаг нуля. Он устанавливается в 1, если результат выполнения команды равен нулю.
SF (Signum Flag)	Флаг знака. Копирует старший бит результата выполнения команды.
OF (Overflow Flag)	Флаг переполнения.
Флаги состояний	
TF (Trap/Trace Flag)	Флаг трассировки. Обеспечивает возможность работы процессора в пошаговом режиме. Например, программа-отладчик DOS DEBUG устанавливает данный флаг так, что возможно пошаговое выполнение каждой команды для проверки изменения содержимого регистров и памяти.
IF (Interrupt Flag)	Флаг внешних прерываний. Если IF=1, прерывания разрешаются, а если IF=0 – блокируются.
DF (Directory Flag)	Флаг направления. Устанавливает направление обработки строк (данных в памяти, превышающих длину одного слова).

При программировании на ассемблере наиболее часто используются флаги OF, SF, ZF и CF для арифметических операций и операций сравнения, а флаг DF — для обозначения направления в операциях над строками. В последующих главах при описании конкретных команд будет дана более подробная информация об этих флагах.

3.3.5. Регистр указателя стека

Регистр SP (Stack pointer) используется при работе со стеком. Стек — это область памяти, организованная для хранения и извлечения данных по принципу "Первым зашел, последним вышел". Регистр SP всегда показывает на вершину стека, т.е. на смещение последнего элемента в стеке, или в сочетании <SS:SP> — на адрес этого элемента. Стек

широко используется для промежуточного хранения данных и адресов, для передачи параметров в процедуры. Регистр SP может использоваться в арифметических или логических командах. Но вообще это НЕЖЕЛАТЕЛЬНО, лучше использовать специальные команды push и pop, которые корректно работают с вершиной стека — см. п. 5.1.4.

3.3.6. Регистры индексов

Регистр SI (Source Index). Этот регистр является индексом источника, обычно он используется в паре <DS:SI> для выполнения операций над строками.

Регистр DI (Destination Index). Этот регистр является индексом назначения (приемника). Пара <ES:DI> применяется также для строковых операций.

Оба индексных регистра можно применять и самостоятельно: для расширенной индексной адресации и для использования в операциях сложения и вычитания.

3.3.7. Регистр базового указателя

Регистр BP (Base Pointer) — указатель базы, облегчает доступ к параметрам: данным и адресам, переданным через стек. Может использоваться для расширенной индексной адресации и в операциях сложения и вычитания.



ЗАМЕЧАНИЕ

При стыковке с модулями, написанными на алгоритмических языках, этот регистр в операциях сложения и вычитания лучше НЕ применять, поскольку он широко используется по своему прямому назначению соответствующими компиляторами.

3.4. Оперативная память и операционные системы IBM PC

Оперативная память (ОЗУ, RAM) физически выполнена в виде микросхем и предназначена для временного хранения программ и данных. Оперативную память можно представить в виде последовательности ячеек, каждая из которых имеет свой номер, который называется *адресом*. С появлением языка С возникло понятие *модель памяти: tiny, small, medium, compact, large, huge*. Это оказалось удобным и для программистов на Ассемблере. Модель памяти начала поддерживаться и компиляторами с Ассемблера, начиная с TASM-3.x.

3.4.1. Сегментная организация памяти

В архитектуру процессора i8086/i8088 была заложена идея *сегментной организации памяти*, которая сохранилась с появлением новых семейств процессоров. Изменялись только адресуемые размеры сегментов, что поддерживалось и новыми операционными системами. Например, в операционной системе MS DOS размер сегмента НЕ мог превышать 64 Кб, поскольку эта операционная система базировалась на 16-разрядных процессорах семейства i8086/i8088/i286, и базовое смещение, которое хранится в регистре IP (для сегмента кода) или в регистре SP (для сегмента стека), НЕ могло превышать величины FFFFh ($2^{16} - 1 = 65535$) — см. рис. 3.3.

При загрузке выполняемой программы (Exe-файла) в память операционная система MS DOS инициализирует, как минимум, три сегментных регистра: CS, DS, SS. Все сегменты могут использовать различные области памяти, могут частично или полностью совпадать (перекрываться). Это зависит как от используемой *модели памяти* — см. табл. 3.3, так и от конкретной задачи. **Обязательным для выполняемой программы является, конечно, сегмент кода.**

Сегментный регистр	**Содержимое сегмента**	**Адрес (Hex)**	**Пояснение**
		00000	Начало оперативной памяти (ОЗУ)
. . .			
CS	Команда	**3DA00**	Начало сегмента кода
. . .			
	Команда	**CS+IP**	Исполнительный (физический) адрес адресуемой команды
. . .			
	Команда	CS+FFFF	Конец сегмента кода (maximum)
. . .			
	Данное	SS-FFFF	Конец сегмента стека (maximum)
. . .			
	Данное	**SS-SP**	Вершина стека (исполнительный адрес)
. . .			
SS	Данное	**78900**	Начало сегмента стека
. . .			
DS	Данное	**89000**	Начало сегмента данных
. . .			
A	Db ?	**DS+a**	Исполнительный (физический) адрес для адресуемой переменной a
. . .			
	Данное	DS+FFFF	Конец сегмента данных (maximum)
. . .			
ES	Данное	**90000**	Начало ДОПОЛНИТЕЛЬНОГО сегмента данных
. . .			
b	Dw ?	**ES+b**	Исполнительный (физический) адрес для адресуемой переменной b
. . .			
	Данное	ES+FFFF	Конец ДОПОЛНИТЕЛЬНОГО сегмента данных (maximum)
. . .			
		FFFFF	Конец ОЗУ

РИС. 3.3. *Структура сегментной организации памяти процессоров i8086/i8088/i286 в операци системе MS DOS.*

На рис. 3.3 **адреса взяты произвольно** из области памяти для пользовательских программ для MS DOS. Операционные системы Windows существенно расширяют адресуемую область ОЗУ и работают в так называемом *защищенном* (многозадачном) режиме.

Для любителей познакомиться с общей схемой распределения памяти в MS DOS — см Прил. 6.

3.4.2. Модели памяти

Размер адресуемой памяти ОЗУ ограничен возможностями *адресной шины процессора* Например, для процессоров семейства i8086/8088 адресная шина имела 20 разрядов (см табл. 3.1). Это значит, что максимально возможный адрес равен FFFFF ($2^{20} - 1 = 1048575$) т.е. 1 Мб — см. рис. 3.3. Для процессоров семейства i286 адресная шина имела уже 24 разряда (см. табл. 3.1). Таким образом, максимально возможный адрес равен FFFFFF ($2^{24} - 1 = 16777215$), т.е. уже 16 Мб.

Предельно допустимые адреса для других процессоров семейства IBM PC вы можете вычислить сами по аналогии, исходя из данных табл. 3.1.

Таблица 3.3. Модели памяти.

Модель	Количество и размер сегментов		Тип указателя	
	Code	Data	Code	Data
MS DOS, Win16				
Tiny	ОДИН, <=64Kb		Near	
Small	ОДИН, <=64Kb	ОДИН, <=64Kb	Near	Near
Medium	НЕСКОЛЬКО, <=64Kb	ОДИН, <=64Kb	Far	Near
Compact	ОДИН, <=64Kb	НЕСКОЛЬКО, <=64Kb	Near	Far
Large	НЕСКОЛЬКО, <=64Kb	НЕСКОЛЬКО, <=64Kb	Far	Far
Huge	НЕСКОЛЬКО, >64Kb	НЕСКОЛЬКО, >64Kb	Huge	Huge
Win32 (начиная с i386)				
Flat	Ограничено возможностями процессора и материнской платы	Ограничено возможностями процессора и материнской платы	Flat	Flat

Модель Tiny используется для создания исполняемой программы в виде com-файла. В этой модели есть только сегмент кода. Данные хранятся внутри последнего. Остальные модели используются для получения exe-файла. **Модель Small** состоит из сегмента кода и сегмента данных. А вот ОЧЕНЬ распространенная **модель Large** в отличие от модели

Small может работать с НЕСКОЛЬКИМИ сегментами кода и данных, но каждый из них НЕ должен превышать 64 Кб. Делается это с помощью организации дальних (FAR) вызовов. **Модели Medium** и **Compact**, как видно из табл. 3.3 занимают промежуточное положение. Более подробно с ближними (NEAR) и FAR вызовами мы разберемся позже, когда будем знакомиться с командами передачи управления — см. главу 9.

Модель Huge была введена в язык С, для того чтобы преодолеть ограничение длины сегмента в 64 Кб (касалось это в то время, в основном, данных). Например, как работать с достаточно большим массивом целых чисел (или символов):

```
char far mas [70000];         // Ошибка!!!
char huge mas  [70000];       // Нормально!
```

С появлением 32-разрядных операционных систем Windows9x/NT4 эта проблема с максимальным размером сегмента в 64 Кб была снята — возникла **модель** *сплошной памяти* **Flat**.

3.4.3. Формирование исполнительного адреса в реальном режиме

В программах НЕ используются физические адреса. Процессоры семейства x86 этого НЕ допускают. **Должен применяться хотя бы один уровень непрямой адресации**. Реальный физический адрес генерируется путем комбинации двух или более чисел. В результате существуют понятия *логического адреса* (или *виртуального*), *линейного* и *физического*. Рассмотрим самую простую и первую по счету концепцию вычисления физических адресов в *реальном режиме*. В этом режиме оказывается любой процессор семейства x86 сразу после включения питания, и он остается в нем, если мы работаем в MS DOS. Операционная система Windows переводит процессор (начиная с модели i386) в *защищенный режим* — см. п. 12.3.2.

Как же процессор узнает, по какому адресу расположена нужная для выполнения команда? Как мы уже упоминали, для этого есть содержимое двух регистров: <CS:IP>. Поскольку сегменты жестко НЕ привязываются к определенным адресам памяти и могут полностью или частично перекрываться, содержимое регистра CS будет разным при разных загрузках исполняемого файла. А вот смещение для каждой команды, в общем-то, величина постоянная и вы можете всегда ее увидеть в lst-файле — см. рис. 2.14, поле 2.

Чтобы повысить эффективность выполнения программ в MS DOS на древних процессорах семейства i8086/i8088/i286, разработчики приняли решение, что адрес сегмента должен быть выравнен на границу *параграфа* (1 параграф = 16 байтам). Таким образом, в **HEX-коде адрес сегмента должен заканчиваться на ноль**. Вернемся к рис. 3.3. Здесь в качестве примера было ПРОИЗВОЛЬНО принято, что содержимое регистра сегмента кода **<CS>=3DA0h** (этот адрес должен быть кратен 16!). Теперь нужно знать содержимое регистра IP (смещение для СЛЕДУЮЩЕЙ команды). Опять же произвольно, но в 16-разрядном формате примем **<IP>=11FAh**.

У нас есть ПОЧТИ вся информация, чтобы вычислить, где ФИЗИЧЕСКИ в ОЗУ находится наша команда. Теперь нужно знать, с каким семейством процессоров мы работаем и разрядность их адресной шины.

Вариант 1. Процессоры i8086/8088. Адресная шина 20-разрядная и, соответственно, как мы выяснили, можно адресовать уже максимум 1 Мб информации. В регистре CS находится 16-разрядное значение. Сделаем его 20-разрядным. Для этого процессор делает любимую команду СДВИГ ВЛЕВО на 4 бита (один HEX-разряд) содержимого регистра CS, а затем складывает полученное с IP:

```
  3DA00h
+  11FAh
 -------
  3EBFAh
```

Мы получили ФИЗИЧЕСКИЙ (исполнительный) адрес в ОЗУ нашей команды.

Вариант 2 (гипотетический). Процессор i286. Адресная шина у этого процессора 24-разрядная и, соответственно, как мы выяснили, можно адресовать уже максимум 16 Мб информации. В регистре CS по-прежнему находится 16-разрядное значение. Сделаем его 24-разрядным. Для этого процессор делает любимую команду СДВИГ ВЛЕВО на 8 битов (два HEX-разряда) содержимого регистра CS, а затем складывает полученное с IP:

```
  3DA000h
+   11FAh
 --------
  3DB1FAh
```

Мы получили ФИЗИЧЕСКИЙ (исполнительный) адрес нашей команды в 24-разрядном коде. Но вообще-то для MS DOS адресация таким образом 16 Мб RAM — это уже НЕПОСИЛЬНАЯ ноша, а для самого процессора — очень НЕЭФФЕКТИВНО, поэтому и появляются в семействе i286 элементы защищенного режима — см. п. 11.1.1.

Вычисление физического адреса ОЗУ при работе со стеком ПОЛНОСТЬЮ аналогично приведенным выше рассуждениям, но с учетом того, что **вершина стека растет в сторону меньших адресов**. Поэтому нужно из сдвинутого влево на соответствующее число разрядов регистра SS **вычесть содержимое регистра SP**, что и показано схематически на рис.3.3.

При работе с данными в сегменте данных уже на этапе компиляции (или компоновки-сборки) известно смещение для всех адресуемых данных. Идея получения исполнительного адреса та же, что и для кодового сегмента.



Основные директивы IBM PC

Трудно по-своему выразить общеизвестные истины.

Квинт Гораций Флакк (65-8 гг. до н.э.)

4.1. Преимущества и недостатки изучения языка Ассемблера с использованием известных алгоритмических языков Pascal и C/C++

Язык Ассемблера можно изучать по-разному. Можно начинать изучать язык сам по себе, а за результатами следить через отладчик — это **обычный, классический путь** (см. п. 5.1.1.3).

Можно изучать Ассемблер, делая полноценные ассемблерные модули (но сначала БЕЗ ввода-вывода) и используя в качестве помощников уже известные алгоритмические языки, например, такие, как **Pascal** или **C/C++**. В данной книге в основном применен именно этот, ВТОРОЙ, подход. Многолетний и многоассемблерный опыт преподавания убедил автора этой книги в его целесообразности. Используя знакомый алгоритмический язык в качестве **помощника** и не вникая сначала в особенности ввода-вывода на Ассемблере и организации исполняемых COM- и EXE-файлов, можно начать с простых команд типа **a+b** (а в Ассемблере, да и в любом алгоритмическом языке ГРАМОТНОЕ решение этой на вид простой задачки дело совсем НЕПРОСТОЕ!). А освоившись с азами программирования на Ассемблере, мы закончим программами на чистом Ассемблере с использованием прерываний и прочих прибамбасов.

Достоинства этого подхода, по мнению автора, заключаются в следующем:

- Ассемблер осваивается постепенно, от простого к сложному.
- Готовые модули на Ассемблере пишутся сразу по мере изучения команд.
- Углубляются знания по уже более-менее освоенному алгоритмическому языку.

- Используются нормальные и уже знакомые операторы ввода-вывода.
- Используется родной для освоенной среды программирования отладчик (интегрированный отладчик). Хотя никто НЕ запрещает использовать и более крутые отладчики.
- Можно подсмотреть (например, в C/C++), как данную задачу решает компилятор и получить чувство глубокого морального удовлетворения, что начинающий программист решает чаще лучше и проще, чем компилятор.

Недостатки:

- Сразу начинаем работать с разноязычными модулями программ (хотя, возможно, это можно отнести к достоинствам).
- Нужно хорошо разбираться в вопросах стыковки (интерфейса) разноязычных модулей — этот материал, как в литературе, так и в документации изложен достаточно путано, а чаще всего в книгах по Ассемблеру НЕ излагается вовсе.

Именно по этому пути мы и пойдем с самого начала (см. п. 5.1.1.2). Таким образом, различные тонкости языка Ассемблера будут раскрываться постепенно, по мере усвоения материала и появления опыта, а значит, и уверенности в своих силах.

Есть еще и ТРЕТИЙ путь — использование *встроенного Ассемблера*. Этот метод мы тоже будем использовать *по мере необходимости* и там, где это *целесообразно*. Но встроенный Ассемблер лишен прелести настоящего Ассемблера: он ОЧЕНЬ зависит от среды его реализации и МНОГИХ команд в нем может просто НЕ быть (особенно это касается **Borland/Turbo Pascal**). Кроме того, зачастую встроенный Ассемблер скрывает (маскирует) КРУПНЫЕ огрехи, которые немедленно возникают в настоящем Ассемблере.

4.2. Разница между директивами и командами Ассемблера

Итак, из п. 2.4.1 мы уже знаем, что в языке Ассемблера существуют команды и директивы, формат которых практически одинаков — см. рис. 2.12. В *командах* **Имя** интерпретируется как **метка**, поэтому за ней всегда ставится символ *двоеточия* ':'. Для Ассемблера в качестве *имени* (*идентификатора*) допускаются следующие символы:

- Латинские буквы от A(a) до Z(z) — РЕГИСТР БЕЗРАЗЛИЧЕН. Ассемблер сам по себе НЕ различает ПРОПИСНЫЕ и строчные буквы. **Но при использовании ассемблерных модулей в программе на алгоритмическом языке, *чувствительном* к регистру букв, например, в C/C++, буквы в разных регистрах будут разными.**
- Цифры от 0 до 9.
- Специальные символы: ? . @ _ $.

Имя в Ассемблере может начинаться с любого символа, кроме цифры. Если имя содержит символ точки '.', то он должен быть ПЕРВЫМ символом. Имя НЕ может быть зарезервированным в Ассемблере словом (имя машинной команды или директивы).

С некоторыми директивами мы с вами уже успели и поработать — см. примеры 2.9 и 2.10.

При ассемблировании между директивами и командами существует одно принципиальное различие. *Директивы* (*псевдооператоры, псевдокоманды*) управляют работой *компилятора* или *компоновщика*, а НЕ микропроцессора. Они используются, например, для сообщения *компилятору*, какие константы и переменные применяются в программе и какие

имена мы им дали, какой сегмент является кодовым, а какой — сегментом данных или стека, в каком формате выводить листинг исходного кода программы и прочее. Большинство директив НЕ генерирует машинных команд (объектный код).

Команда Ассемблера всегда генерирует машинный код.

Директивы имеют **разный** синтаксис в режимах *MASM* (поддерживается компиляторами Microsoft Assembler — **masm** и Borland (Turbo) Assembler — **tasm**) и *Ideal* (поддерживается компилятором **tasm**) — см. приложение 5.

При описании *синтаксиса директив и команд* обычно используют специальный язык, известный всем профессиональным программистам, — **язык Бэкуса-Наура** (названный так в честь этих двух достойных ученых). Мы будем использовать его упрощенный вариант:

- *Терминальные элементы* языка (название директив и команд, разделительные знаки) будем выделять **жирным** шрифтом.
- *Нетерминальные элементы* (название параметров, операндов и других атрибутов) выделим *курсивом*.
- Комментарии будем писать обычным шрифтом.
- *Необязательные элементы* заключаются в квадратные скобки []. В синтаксисе языка Ассемблера есть и **терминальный элемент квадратные скобки []** — в этом случае это будет особо ПОДЧЕРКНУТО.
- *Повторяющиеся* элементы синтаксической структуры описываются многоточием (...).

Директив достаточно много. Практически каждая версия компилятора что-то из директив добавляет или немного изменяет синтаксис. Для определенности мы в данной главе остановимся на синтаксисе *основных директив* в более универсальном формате *MASM* для компиляторов **masm-6.12** (или выше) и **tasm-3.1** (или выше). Кроме того, известные компиляторы с языка программирования С/С++ (**Borland C++, Visual C++**) выдают ассемблерный листинг именно в формате *MASM*. И очень скоро мы научимся его читать...

4.3. Описание сегмента — директива SEGMENT

Из п. 2.4.3 мы знаем, что любые ассемблерные программы содержат, по крайней мере, один сегмент — сегмент кода. В некоторых программах используется сегмент для стековой памяти и сегмент данных (основной и дополнительный) для определения данных.

Универсальная директива для описания сегмента имеет следующий формат:

```
имяС    SEGMENT    [параметры] ; начало СЕГМЕНТА имяС
              .
              .
              .
имяС    ENDS    ; конец СЕГМЕНТА имяС
```

Имя сегмента (*имяС*) должно обязательно присутствовать, быть уникальным и соответствовать соглашениям для имен в Ассемблере или в другом алгоритмическом языке, для стыковки с которым делается ассемблерный модуль. Например, при стыковке Ассемблера с Turbo/Borland Pascal *имяС* должно быть СТРОГО определенным:

- для сегмента кода *CSEG* или *CODE*;
- для сегмента данных *DSEG* или *DATA*;
- для сегмента стека *STACK*.

Директива **ENDS** обозначает конец сегмента. Обе директивы **SEGMENT** и **ENDS** должны иметь одинаковые имена *имяС*.

В одном модуле можно открывать и закрывать сегмент с одним и тем же именем *имяС* несколько раз, но пересекаться (вкладываться друг в друга) разные сегменты НЕ должны. Компилятор просматривает ассемблерный модуль и объединяет вместе все части сегментов с одинаковым именем в том порядке, в каком он их обнаруживает (сверху-вниз).

Директива **SEGMENT** может содержать три **основных** типа *необязательных* параметров, определяющих выравнивание (*align*), объединение (*combine*) и класс (*'class'*), между которыми должен быть хотя бы один пробел в качестве разделителя. Параметры имеют смысл при разработке БОЛЬШИХ ассемблерных программ.

1. **Выравнивание (*align*).** Этот параметр сообщает *компоновщику*, чтобы он разместил данный сегмент, начиная с указанной границы. Это может быть ОЧЕНЬ важно, т.к. при правильном выравнивании данные загружаются процессором с большей скоростью. При попытке сосчитать невыравненные данные процессор сделает одно из двух: либо возбудит исключение, либо сосчитает их в несколько приемов (конечно, при этом быстродействие программы снизится) [Л7-8].

Таблица 4.1. Параметр выравнивания (align)

Параметр	Значение
BYTE	Выравнивание НЕ выполняется. Сегмент размещается, начиная со следующего байта.
WORD	Начало сегмента выравнивается на границу слова (четный адрес, кратный 2).
DWORD	Начало сегмента выравнивается на границу двойного слова (четный адрес, кратный 4).
PARA	**Начало сегмента выравнивается на границу параграфа (четный адрес, кратный 16 - см. п. 3.4.3). Это значение принято по умолчанию.**
PAGE	Начало сегмента выравнивается на границу страницы (четный адрес, кратный 256).
MEMPAGE	Начало сегмента выравнивается на границу страницы памяти (четный адрес, кратный 4K).

2. **Объединение (*combine*).** Настоящий элемент предназначен для указания *компоновщику*, каким образом объединять сегменты, находящиеся в разных модулях и имеющие одинаковые имена, или как соединить данный сегмент с другими сегментами в процессе компоновки после ассемблирования.

3. **Класс (*'class'*).** Данный элемент, заключенный в апострофы, используется для группирования сегментов при компоновке. Компоновщик группирует вместе все сегменты с ОДИНАКОВЫМ классом.

Таблица 4.2. Параметр объединения (*combine*)

Параметр	Значение
PRIVATE	Сегмент НЕ будет объединяться с сегментами с тем же именем, находящимися в других модулях. Это значение принято по умолчанию.
PUBLIC	Сегмент будет объединяться с сегментами с тем же именем, находящимися в других модулях или в том же модуле, в один сегмент.
MEMORY	Параметр подобен PUBLIC, но для *сегмента стека.*
COMMON	Размещает данный сегмент и все сегменты с тем же именем по ОДНОМУ и тому же адресу. Получаются перекрывающиеся сегменты (*overlay*), занимающие разделяемую область памяти (*shared memory*).
VIRTUAL	Описывает сегмент специального вида, который должен объявляться ВНУТРИ другого сегмента (общая область памяти).
AT xxx	Устанавливает сегмент по абсолютному адресу параграфа xxx для получения доступа по идентификаторам к фиксированным адресам памяти (например, видеобуфер, таблица векторов прерываний – см. прил.6).

Например, *сегмент стека*, в котором зарезервировано 100*8 = 800 байтов (со словом My Stack — 8 символов) может быть описан следующим образом:

```
SStack  SEGMENT   PARA  PUBLIC  'Stack'
        DB        100 dup ('My Stack')
SStack  ENDS
```

4.4. Директива группирования сегментов Group

Эта директива используется для объединения сегментов в группу. Она имеет следующий формат:

имяG **GROUP** *имяC1*[, *имяC2...*]

Группа позволяет осуществлять доступ к данным из всех сегментов, которые находятся в ней, с помощью одной загрузки адреса группы в сегментный регистр. Этим широко пользуются компиляторы C/C++.

4.5. Директива Assume

Эта директива сообщает *компилятору*, что указанный в ней сегментный регистр необходимо связать с именем сегмента или группы. Она имеет следующий формат:

ASSUME *сегм_регистр1:имя1*[, *сегм_регистр2: имя2...*]

В качестве сегментных регистров для базового Ассемблера принимаются уже известные нам регистры: CS, DS, ES или SS.

Для ОТМЕНЫ назначения для данного сегментного регистра используется ДРУГОЙ формат этой директивы:

ASSUME *сегм_регистр1*:**NOTHING**[,*сегм_регистр2*: **NOTHING** ...]

Чаще всего эта директива используется в начале модуля на Ассемблере.

Рассмотрим фрагмент листинга ассемблерного файла, который был получен компилятором **Borland C++ 5.02**:

```
_TEXT   segment byte public 'CODE'
_TEXT   ends
DGROUP  group _DATA,_BSS
             assume    cs:_TEXT,ds:DGROUP
_DATA   segment word public 'DATA'
             ......................................
_DATA   ends
_BSS         segment word public 'BSS'
             ......................................
_BSS         ends
_TEXT   segment byte public 'CODE'
        assume    cs:_TEXT,ds:DGROUP
             ......................................
_TEXT   ends
_DATA   segment word public 'DATA'
             ......................................
_DATA   ends
_TEXT   segment byte public 'CODE'
_TEXT   ends
        end
```

Ну, как? Все понятно? Вот мы и начинаем немного понимать, что же делают компиляторы с алгоритмических языков...

4.6. Стандартные модели памяти

Современные версии компиляторов с Ассемблера позволяют упростить описание сегментов с помощью использования стандартных моделей памяти — см. п. 3.4.2. В этом случае директивы **SEGMENT**, **ENDS** и **ASSUME** становятся НЕНУЖНЫМИ.

4.6.1. Директива MODEL

Директива **MODEL** позволяет задавать в ассемблерной программе одну из нескольких стандартных моделей сегментации памяти (*модель_памяти)* — см. табл. 3.3. Приведем ее *упрощенный* формат, которым мы будем пользоваться в дальнейшем:

.MODEL *модель_памяти* [, *язык*]

Параметр *язык* позволяет немного упростить вопросы интерфейса (стыковки) модуля на Ассемблере с общепринятыми алгоритмическими языками. Если вы их используете, то параметр *язык* должен быть равен одному из терминальных элементов: **C**, **CPP**, **BASIC**, **PASCAL**, **FORTRAN**, **PROLOG**. Если этот параметр НЕ указан, он считается **NOLANGUAGE**. Например:

.MODEL **Large , C**

.MODEL **Small , Pascal**

4.6.2. Директивы упрощенного описания сегментов

Определившись с используемой моделью, можно использовать упрощенные директивы описания **основных** сегментов (регистр букв НЕСУЩЕСТВЕНЕН!):

.Code ; формат MASM
CODESEG ; формат Ideal — см. Прил. 5.

.DATA ; формат MASM
DATASEG ; формат Ideal

.STACK ; формат MASM
STACK ; формат Ideal

4.7. Описание процедур

Модуль на Ассемблере, как и модули на алгоритмических языках, обычно состоит из процедур. Для описания процедур используются две директивы. Визуально они похожи на соответствующие директивы описания сегмента — см. п. 4.2:

```
имяP    PROC    [параметры] ; начало процедуры имяP
        ..........................................
        RET   ; КОМАНДА возврата в точку вызова процедуры
        ..........................................
имяP    ENDP    ; конец процедуры имяP
```

У этой директивы достаточно много *параметров*. С ними мы будем знакомиться постепенно, по мере необходимости.

Обратите внимание на ОБЯЗАТЕЛЬНУЮ команду **RET**. Она может быть в любом нужном месте процедуры и НЕ единственная. Если ее в процедуре НЕ будет, то ассемблерная программа НЕ сможет нормально работать — возникнет зависание.

4.8. Описание внешних ссылок

Как было заявлено в п. 4.1, мы будем использовать алгоритмические языки Pascal и C/C++ в качестве помощников при изучении Ассемблера. Таким образом, сразу начинаем работать с РАЗНЫМИ модулями, да еще и на разных языках! Поэтому нам не миновать ВНЕШНИХ ссылок. Что это такое? Это — использование в одном модуле имен, описанных в других модулях.

4.8.1. Директива описания общих имен PUBLIC

PUBLIC [*язык*] *имя1*[,[*язык*] *имя2*...]

Эта директива указывает компилятору и компоновщику, что данное *имя* (его адрес) должно быть доступно для других программ. Имена могут быть метками, переменными или именами подпрограмм.

Например, если мы хотим использовать в языке C/C++ функцию с именем Prim, ре ализованную в Ассемблере, то она в ассемблерном модуле должна быть описана следую щим образом:

```
        Public    C  Prim
Prim    Proc
        .....................................
Prim    EndP
```



ОБРАТИТЕ ВНИМАНИЕ

Язык C/C++ различает регистр букв в именах. Это же должен делать и Ассемблер, для чего служит **специальный ключ**, определяющий чувствительность Ассемблера к РЕГИСТРУ выбора символов: ml=all (все символы), mx=globals (только глобальные), mu=none (символы к регистру НЕ чувствительны — принято по умолчанию):

TASM имя.**asm**[**/ml**]

TASM имя.**asm**[**/mx**]

TASM имя.**asm**[**/mu**]

В нашем случае ассемблерный модуль должен быть откомпилирован ОБЯЗАТЕЛЬН(с ключом **/ml** или **/mx**. Иначе *компоновщик C/C++* НЕ сможет подключить ассемблерны модуль. Если эти рассуждения вам пока НЕПОНЯТНЫ, НЕ зацикливайтесь, — мы к это му еще вернемся на КОНКРЕТНЫХ примерах.

4.8.2. Директива описания внешних имен EXTRN

EXTRN *имя1:тип1*[, *имя2:тип2...*]

Эта директива указывает компилятору и компоновщику, что данное *имя* имеет опре деленный *тип*, его предполагается использовать в данном ассемблерном модуле, но **па мять для него выделена в другом модуле**. Параметр *тип* может принимать следующие зна чения: **ABS** (для констант), **BYTE, WORD, DWORD, QWORD, TBYTE** (см соответствующий столбец в табл. 2.2), **FAR, NEAR**.

Например, в модуле на алгоритмическом языке **Pascal (Borland/Turbo Pascal-5.5/6.0, 7.0x)** мы описали следующие **глобальные** переменные:

Var a, b, c : **Integer**;
    X   : **LongInt**;

А использовать их собираемся в ассемблерном модуле. В этом случае директива буде иметь вид:

```
Extrn   a:Word, b:Word, c:Word, x:Dword
```



ОБРАТИТЕ ВНИМАНИЕ

Язык Pascal НЕ различает регистр букв в именах. А директива EXTRN содержит только ОДНУ гласную букву — начинающие программисты часто на этом спотыкаются... Будьте ВНИМАТЕЛЬНЫ!

Вопросы

1. Что изменится в директиве **EXTRN**, если мы опишем указанные выше переменные на алгоритмическом языке **Object Pascal (Borland Delphi, Win32 Console)**?
2. Что изменится в директиве **EXTRN**, если мы опишем указанные выше переменные на алгоритмическом языке **C++ (Borland C++ 4.5/5.02, EasyWin)**?
3. Что изменится в директиве **EXTRN**, если мы опишем указанные выше переменные на алгоритмическом языке **C++ (Borland C++ 4.5/5.02, MS DOS Standard)**?
4. Что изменится в директиве **EXTRN**, если мы опишем указанные выше переменные на алгоритмическом языке **C++ (Borland C++ 4.5/5.02, Win32 Console)**?
5. Что изменится в директиве **EXTRN**, если мы опишем указанные выше переменные на алгоритмическом языке **C++ (Microsoft Visual C++ 5.0/6.0, Win32 Console)**?
6. Что изменится в директиве **EXTRN**, если мы опишем указанные выше переменные на алгоритмическом языке **Object Pascal (Borland Delphi, Win32 Application)**?
7. Что изменится в директиве **EXTRN**, если мы опишем указанные выше переменные на алгоритмическом языке **C++ (Microsoft Visual C++ 5.0/6.0, Win32 Application)**?

Если вы правильно ответили на ВСЕ вопросы, значит, вы ХОРОШО УСВОИЛИ главу 2 (п. 2.2) — поздравляю!!! Если НЕТ, не беда — все у вас еще впереди (просто пока маловато опыта).



Основные команды целочисленной арифметики IBM PC XT/AT

Тяжело в учении, легко в бою.

А.В. Суворов (1730–1800 гг.)

Теперь пришло время познакомиться с командами. Ассемблер может работать с достаточно широким диапазоном данных. Начнем мы с простейших **базовых команд** целочисленной арифметики. Рассматриваемые в этой главе команды будут работать со знаковыми или беззнаковыми данными длиной 8 или 16 бит, с которыми мы уже познакомились — см. п. 2.2. Некоторые команды смогут обрабатывать и 32-разрядные данные (команды пересылки, команды сложения и вычитания).

Основное большинство этих команд НЕ различает знаковые и беззнаковые данные — это прерогатива человека. Мы с вами видели, что за знак "отвечает" самый старший бит числа в его внутреннем (машинном) представлении — см. п. 2.2.2. Учитывать ли его как знак или как информационную часть числа решает человек. Для начинающих программистов это достаточно "скользкий" момент — постарайтесь его понять, а я вам буду помогать...

5.1. Команды пересылки и обмена информацией

Это самые простые и распространенные команды. На самом деле команд пересылок в Ассемблере гораздо больше, но мы с вами договорились — начнем с базовых команд. Поняв принцип их действия, вы "породнитесь" с остальными командами.

Еще мы знаем, что команды Ассемблера содержат *мнемокод* (*мнемонический код* — см. п. 2.4.1), призванный облегчить понимание человеком данной команды. Для русскоязычного читателя, не знающего английский язык, это может служить дополнительным источником мучений, но тут уж ничего не поделаешь... Английский язык (de facto) стал международным языком общения программистов. Поэтому я буду обязательно раскрывать мнемонику каждой команды на языке оригинала (английском) с переводом на русский язык. Авось и у вас появится необходимый словарный запас, и вы сможете понимать систему помощи HELP. Ну а тем, кто знает английский язык, и карты в руки...

5.1.1. Команда пересылки MOV

Мнемокод этой команды получен в результате сокращения такого предложения: MOVe operand to/from system registers — Пересылка операнда в/из системных регистров. Команда эта содержит **два операнда** и имеет следующий формат (**синтаксис**):

MOV *Destination, Source*
MOV *Приемник, Источник*

Это ОЧЕНЬ распространенный формат команд и, если команда содержит два операнда, они почти всегда интерпретируются именно так: первый операнд — приемник, а второй — источник.

В данном конкретном случае информация из источника пересылается (КОПИРУЕТСЯ) в приемник. Эта команда в **самой простой** своей интерпретации соответствует **оператору присваивания** (содержимое источника <*Источник*> копируется в приемник — см. табл. 5.1):

Приемник := <*Источник*>

Вообще же, как мы убедимся в дальнейшем, возможности ее шире.

ДЛИНА ОБОИХ ОПЕРАНДОВ должна быть ОДИНАКОВОЙ. Если же она разная, используется директива указания типа:

тип **PTR** [*выражение*]

Например, **BYTE PTR a+2**

Таблица 5.1. Возможные сочетания операндов для команды MOV.

Приемник (Destination)		Источник (Source)		Простейший пример использования
Регистр (Register)	R8	Регистр	R8	MOV AL,DH
	R16		R16	MOV AX,DX
	R8	Память	Mem8	k DB 10 .... MOV CL, k
	R16		Mem16	EXTRN k1:WORD .... MOV SI, k1
	R8	Константа (Immediate)	Im8	MOV BL,15
	R16		Im16	MOV AX,1578
Память (Memory)	Mem8	Регистр	R8	m DB ? .... MOV m, CL
	Mem16		R16	EXTRN k2:WORD .... MOV k2, DI
	Mem8	Константа	Im8	EXTRN S:BYTE .... MOV S,0
	Mem16		Im16	y DW ? .... MOV y, 12345

5.1.1.1. Исключения

У этой простой на первый взгляд команды есть исключения:

1. В качестве приемника НЕ может быть регистр **CS**.
2. Нельзя записать команду ПРЯМОЙ инициализации сегмента данных:

```
DSEG    SEGMENT
..........................
        Mov DS, Dseg
```

В этой ситуации команду MOV можно расписать на две:

```
        Mov AX,Dseg
        Mov DS,AX
```

3. Нельзя пересылать СЕГМЕНТНЫЕ регистры:

```
        Mov ES,DS
```

В этой ситуации можно сделать так:

```
        Mov AX,DS
        Mov ES,AX
```

Аналогично нельзя использовать и команду **Mov DS,ES**.

ПРИМЕР 5.1.

Реализуем на Ассемблере простейшую программу присваивания ЦЕЛОЧИСЛЕННЫХ данных длиной 8 и 16 бит. Пусть данное длиной 16 бит будет ЗНАКОВОЕ, а — 8 бит БЕЗЗНАКОВОЕ. Ввод-вывод информации сделаем на Паскале (**Borland/Turbo Pascal**). Программа на Паскале будет ГЛАВНОЙ, из нее мы будем вызывать модуль на Ассемблере. Соглашений по интерфейсу **Pascal+Assembler** мы пока еще НЕ знаем (см. п. 6.1), но, думаю, что листинги этих двух модулей будут понятны — все директивы и команды, надеюсь, вам известны. Проанализируйте их внимательно.

Исходный текст программы mov16.pas

```
Program Mov16;
{----------------------------------------------------------------
            x=a;  a,x : Integer;
            z=b;  b,z : Byte;
             Лекционный пример 5.1.
         CopyRight by Голубь Н.Г., 2001
----------------------------------------------------------------}
{$L Mov_16}   {вызов ASM-модуля Mov_16.obj}
{$f+}{Директива FAR-вызова процедур и функций}
Var
 x,a   : Integer;
 b,z   : Byte;
```

```
{Описание внешней процедуры — она реализована на Ассемблере}
Procedure Mov_16(var x:integer; var z:Byte); {FAR;} external;
{FAR НЕ обязателен — есть директива!}
begin
   Writeln(' Вычислить:     x = a; a,x : Integer;');
   Write('Введите значение a  ');
   Readln(a);
   x := a;
   Writeln(' Вычислить:     z = b; b,z : Byte;');
   Write('Введите значение b  ');
   Readln(b);
   z := b;
   writeln ('ПАСКАЛЬ:  x=',x, ';  z=',z);
   {Делаем вычисление этих переменных на Ассемблере}
   z:=0;
   x:=0;
   mov_16(x,z);
   writeln ('АССЕМБЛЕР: x=',x, ';  z=',z);
   Readln
end.
```

Исходный текст модуля mov_16.asm

```
; mov_16.asm
         title Move for 8 or 16 bits
; CopyRight by Голубь Н.Г., 2001
         .MODEL Large
         .data
;-------  принимаем/передаем значение из/в Паскаль
         ;x,a:Integer
         Extrn    a: WORD, x:word
         ;z,b:byte
         Extrn    b: BYTE, z:BYTE
         .code
         Public   Mov_16
Mov_16  proc     far
         mov      ax,a        ; ax <=== a
         mov      bL,b        ; bL <=== b
         mov      x,ax        ; x <=== <ax>
         mov      z,bL        ; z <=== <bL>
         ret
Mov_16  endp
         end
```

Ассемблер, как вы видите, НИКАК не отреагировал на то, что есть данные знаковые и беззнаковые — ему ВАЖНА **длина** операндов! Реагирует только Паскаль, поскольку именно он ИНТЕРПРЕТИРУЕТ данные на вводе-выводе.

Теперь с этими двумя модулями мы можем и поэкспериментировать...

Чтобы получить результат, нужно сделать следующее:

1) Вызвать среду программирования **Borland/Turbo Pascal**. В редакторе набрать в одном окне файл **mov16.pas**, а в другом — **mov_16.asm** (следите, чтобы имена модулей были РАЗНЫЕ!!! Иначе на этапе компоновки вы рискуете озадачить компоновщик).

2) Откомпилировать файл **mov_16.asm**, вызвав через **Shift-F3** компилятор **TASM** (он входит в стандартную поставку и, если установка **Borland/Turbo Pascal** была сделана *грамотно, операционной системе должны быть известны все пути*). Можно откомпилировать ассемблерный файл и отдельно (лучше сразу с листингом — получим файл **mov_16.lst**) — мы это уже делали (см. п. 2.4.5). Если НЕТ ошибок (что с первого раза бывает ОЧЕНЬ редко), должен получиться ОЧЕНЬ НУЖНЫЙ нам объектный файл **mov_16.obj**. Если есть ошибки, надо их исправить и повторить процесс ассемблирования до победного конца.

3) Запустить, как обычно, на компиляцию и счет наш ГЛАВНЫЙ файл **mov16.pas**.

Ну а дальше по известному сценарию: кто врет? Того и наказывать... Т.е. исправлять и перекомпилировать.



ПРИМЕР 5.2.

Теперь решим более сложную для 16-разрядного программирования задачу. Переслать информацию из области памяти **A** длиной 32 бита. Чтобы удобно было наблюдать и делать выводы, занесем в нашу переменную какой-нибудь ОПРЕДЕЛЕННЫЙ HEX-код. Например, **A=12345678h**.

Один из вариантов решения этой задачи заключается в том, что мы можем воспользоваться ДВУМЯ командами MOV для пересылки информации по 16 бит (младшей и старшей части 32-разрядного числа):

Исходный текст модуля mov32_16.asm

```
        ; mov32_16.asm
; для стыковки с Turbo/Borland Pascal
        title Move for 32 bits
; CopyRight by Голубь Н.Г., 2001
        .MODEL Large
        ;x,a:LongInt
        .data
        Extrn   x:Dword    ; передаем значение в Паскаль
a       DD      12345678h  ; внутренняя переменная a
        .code
        Public  MovL
MovL    proc    far
        mov     ax,WORD PTR a       ; ax <=== мл. часть a
        mov     bx,WORD PTR a+2     ; bx <=== ст. часть a
        mov     WORD PTR x,ax       ; мл. часть x<=== <ax>
        mov     WORD PTR x+2,bx     ; ст. часть x<=== <bx>
        ret
MovL    endp
        end
```



ПОЯСНЕНИЯ

В примере 2.2 было упомянуто, что реально в памяти компьютера числа хранятся "задом наперед". Применительно к нашему случаю — см. рисунок 5.1.

Исходное число	12345678h											
Внутреннее (машинное) представление												
Имя	Биты											
A	15	14	13	...	2	1	0	31	30	...	18	17	16
Смещение в памяти	A							A+2					
Содержимое регистров (ПОСЛЕ пересылки) HEX-код	AX							BX					
	5678							1234					
Интерпретация	Младшая часть числа							Старшая часть числа					

РИС. 5.1. *Расположение в памяти IBM PC числа a=12345678h и результат выполнения команды MOV.*

За содержимым регистров AX и BX очень просто наблюдать в ИНТЕГРИРОВАННОМ отладчике **TURBO Pascal (Borland Pascal** — только **Real Mode)**.

5.1.1.2. Подключение интегрированного отладчика TURBO Pascal-7.0x

1) Создать главный модуль на Паскале по образу и подобию модуля примера 5.1 (файл **Mov32.pas** есть в материалах, прилагаемых к данной книге).
2) Занести в EXE-файл отладочную информацию — **Options/Compiler/Debugging**:
   - **Debug Information**
   - **Local Symbols**
3) Подключить отладчик — **Options/Debugger**:
   - **Integrated**
4) Установить контрольные точки: **F4 — Go to cursor**.
5) Подключить окно CPU: **Debug/Register**.
6) Запустить отладочный режим: **F7 — trace** (с трассировкой подпрограмм), **F8 — Step** (БЕЗ трассировки подпрограмм).

5.1.1.3. Использование отладчика Debug

Для любителей **традиционного подхода** сделаем чисто ассемблерную версию нашей программы (подробности — см. в п. 14.1). Видите ли вы совсем НЕ *маленькую* разницу в ассемблерных файлах? Здесь даже есть команды, которые мы еще и НЕ изучали...

Исходный текст модуля mov32_16.asm

```
          ; mov32_16.asm — законченная программа
          title Move for 32 bits
;CopyRight by Голубь Н.Г., 2001
          ;x,a:LongInt
DSEG     SEGMENT
```

```
a        DD       12345678h
x        dd       ? ; резервируем память
DSEG     EndS
code     SEGMENT PARA 'code'
          ASSUME  CS:code, DS:DSEG
          Public  MovL
MovL     proc     far
;——————— Подготовка НОРМАЛЬНОГО возврата в MS DOS ———
          PUSH     DS
          XOR      DI,DI
          push     DI
;———————— Инициализация сегмента данных ————————
          mov      ax,DSEG
          mov      DS,ax
;————————————————————————————————————————————
          mov      ax,WORD PTR a        ; ax <=== мл. часть a
          mov      bx,WORD PTR a+2      ; bx <=== ст. часть a
          mov      WORD PTR x,ax   ; мл. часть x <=== <ax>
          mov      WORD PTR x+2,bx; ст. часть x <=== <bx>
          ret
MovL     endp
code     EndS
          end  MovL; точка входа в программу ОБЯЗАТЕЛЬНА!
```

Затем, как обычно, откомпилируем этот файл (с листингом):

tasm mov32_16.asm /L

Получим объектный файл **mov32_16.obj**. Теперь мы вызовем компоновщик:

tlink mov32_16.obj

В результате получим исполняемый файл **mov32_16.EXE**.

А теперь вызовем отладчик **Debug**, чтобы посмотреть содержимое регистров. Воспользуемся самыми простыми командами:

- **r (register)** — просмотр регистров;
- **t (trace)** — трассировка;
- **q (quit)** — выход из отладчика.

Вот что он выдает в моей любимой операционной системе **Windows NT 4 SP6** (в окне **MS DOS**):

```
F:\DiaSoft\Examples\Tasm\Part1\5\5_2\Debug>debug MOV32_16.EXE
-r
AX=0000  BX=0000  CX=0028  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0C3D  ES=0C3D  SS=0C4D  CS=0C4E  IP=0000   NV UP EI PL NZ NA PO NC
0C4E:0000 1E           PUSH    DS
-t
AX=0000  BX=0000  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0C3D  ES=0C3D  SS=0C4D  CS=0C4E  IP=0001   NV UP EI PL NZ NA PO NC
0C4E:0001 33FF         XOR     DI,DI
..........................................................................
```

```
-t
AX=0000  BX=0000  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0C40  ES=0C40  SS=0C50  CS=0C51  IP=0003   NV UP EI PL ZR NA PE NC
0C51:0003 57            PUSH    DI
-t
AX=0000  BX=0000  CX=0028  DX=0000  SP=FFFC  BP=0000  SI=0000  DI=0000
DS=0C40  ES=0C40  SS=0C50  CS=0C51  IP=0004   NV UP EI PL ZR NA PE NC
0C51:0004 B8500C        MOV     AX,0C50
-t
AX=0C50  BX=0000  CX=0028  DX=0000  SP=FFFC  BP=0000  SI=0000  DI=0000
DS=0C40  ES=0C40  SS=0C50  CS=0C51  IP=0007   NV UP EI PL ZR NA PE NC
0C51:0007 8ED8          MOV     DS,AX
-t
AX=0C50  BX=0000  CX=0028  DX=0000  SP=FFFC  BP=0000  SI=0000  DI=0000
DS=0C50  ES=0C40  SS=0C50  CS=0C51  IP=0009   NV UP EI PL ZR NA PE NC
0C51:0009 A10000        MOV     AX,[0000]                  DS:0000=5678
-t
AX=5678  BX=0000  CX=0028  DX=0000  SP=FFFC  BP=0000  SI=0000  DI=0000
DS=0C50  ES=0C40  SS=0C50  CS=0C51  IP=000C   NV UP EI PL ZR NA PE NC
0C51:000C 8B1E0200      MOV     BX,[0002]                  DS:0002=1234
-t
AX=5678  BX=1234  CX=0028  DX=0000  SP=FFFC  BP=0000  SI=0000  DI=0000
DS=0C50  ES=0C40  SS=0C50  CS=0C51  IP=0010   NV UP EI PL ZR NA PE NC
0C51:0010 A30400        MOV     [0004],AX                  DS:0004=0000
-t
AX=5678  BX=1234  CX=0028  DX=0000  SP=FFFC  BP=0000  SI=0000  DI=0000
DS=0C50  ES=0C40  SS=0C50  CS=0C51  IP=0013   NV UP EI PL ZR NA PE NC
0C51:0013 891E0600      MOV     [0006],BX                  DS:0006=0000
-t
AX=5678  BX=1234  CX=0028  DX=0000  SP=FFFC  BP=0000  SI=0000  DI=0000
DS=0C50  ES=0C40  SS=0C50  CS=0C51  IP=0017   NV UP EI PL ZR NA PE NC
0C51:0017 CB            RETF
-q
```

Ну как? Что вам понравилось больше? Традиционный отладчик **Debug** — это тоже хорошее дело, если уже есть какой-то опыт... Дерзайте!!!

5.1.2. Команда обмена данными XCHG

Эта команда МЕНЯЕТ МЕСТАМИ содержимое двух операндов (eXCHanGe).

Синтаксис:

XCHG *Приемник*, *Источник*

Таблица 5.2. Возможные сочетания операндов для команды XCHG.

Приемник	Источник	Пример
R8	R8	XCHG BL, BH
R16	R16	XCHG AX, BX
R8	Mem8	K DB 10 .... XCHG CL, k
R16	Mem16	a Dw ? .... XCHG DI, a
Mem8	R8	s DB -100 .... XCHG s, DL
Mem16	R16	ab Dw ? .... XCHG ab, SI

5.1.3. Команда загрузки адреса LEA

Команда загружает в 16-разрядный регистр 16-разрядное смещение (Load Effective Address).

Синтаксис:

LEA *Приемник*, *Источник*

Таблица 5.3. Возможные сочетания операндов для команды LEA.

Приемник	Источник	Пример
R16	Mem16	a Dw ? .... LEA DI, a

Эту же команду (в ПРОСТЕЙШЕМ случае) можно реализовать и через уже известную нам команду **MOV** с использованием директивы **OFFSET**:

```
X       DW      1000
...............................................
        Mov     BX, OFFSET x
```

Но команда LEA более гибко может использовать разные режимы адресации.

? ВОПРОС.

Одинаковы ли по своему результату следующие ниже команды MOV?

```
X       DW   1000
        Mov  BX,  OFFSET x
        Mov  BX,  x
```

Можете уже сами создать ассемблерный файл и проверить с помощью отладчиков. А еще лучше, если вы четко знаете, в чем же тут разница.



ПОДСКАЗКА.

Вспомните, что смещение мы всегда видим в файле листинга — см. поле 2 на рис. 2.14.

5.1.4. Команды работы со стеком

Обычно эти команды работают в паре:

PUSHx — сохранить информацию в стеке;

POPx — восстановить информацию из стека.

Мы знаем (см. п. 3.3.5), что стек заполняется задом наперед, т.е. в сторону меньших адресов. Поэтому все команды **PUSHx** сначала уменьшают значение указателя стека **<SP>=<SP>-2** (регистр **SP** всегда показывает на вершину стека), а затем копируют в стек операнды. Команды **POPx**, наоборот, сначала извлекают нужную информацию в порядке, ОБРАТНОМ записи в стек, а затем изменяется содержимое вершины стека: <SP>=<SP>+2.

5.1.4.1. Команды PUSH и POP

Эти команды запоминают или извлекают 16-разрядный операнд в/из стека.

Синтаксис:

PUSH *Источник*
POP *Приемник*

Таблица 5.4. Допустимые форматы команд PUSH/POP.

Команды	
PUSH R16	POP R16
PUSH Mem16	POP Mem16
PUSH CS	**POP** CS НЕЛЬЗЯ!!!
PUSH DS	POP DS
PUSH ES	POP ES
PUSH SS	POP SS
PUSH DS ES SS	POP SS ES DS
PUSH DX AX BX CX	POP CX BX AX DX



ПРИМЕР 5.3.

Из п. 5.1.1.1 мы знаем, что НЕЛЬЗЯ пересылать СЕГМЕНТНЫЕ регистры:

Mov ES,DS

В этой ситуации можно воспользоваться стеком:

```
PUSH DS
POP  ES
```

Аналогично НЕЛЬЗЯ использовать и команду **Mov DS,ES**, а через стек можно:

```
PUSH ES
POP  DS
```

5.1.4.2. Команды PUSHA и POPA

Команда **PUSHA** запоминает все (ALL) регистры в стеке СТРОГО в следующей последовательности: AX, CX, DX, BX, SP, BP, SI, DI.

Команда **POPA**, наоборот, извлекает все (ALL) эти регистры из стека: DI, SI, BP, SP, BX, DX, CX, AX.

Синтаксис:

PUSHA
POPA

Этим командам операнды НЕ нужны.

5.1.4.3. Команды PUSHF и POPF

Эти команды запоминают (**PUSHF**) или восстанавливают (**POPF**) регистр флагов **FLAGS**, используя стек.

Синтаксис:

PUSHF
POPF

Этим командам операнды тоже НЕ нужны.

 **ПРИМЕЧАНИЕ**

Прямой доступ к регистру флагов НЕВОЗМОЖЕН. Настоящие команды позволяют обойти это ограничение и широко используются в программах обработки прерываний.

5.1.5. Команды загрузки полного указателя LDS, LES и LSS

Эти команды (Load pointer into DS/ES/SS) загружают из 32-разрядной области памяти *Mem32* ПОЛНЫЙ указатель в соответствующую пару <*сегментный_ регистр : регистр*>.

Синтаксис:

LDS *Приемник, Источник*
LES *Приемник, Источник*
LSS *Приемник, Источник*

Содержимое источника		
Область памяти (смещение)	Mem32 (первые 16 бит)	Mem32+2 (вторые 16 бит)
Результат выполнения команды	**Приемник (16-разрядный регистр)**	**Неявный сегментный регистр**
LDS	R16	DS
LES	R16	ES
LSS	R16	SS

РИС. 5.5. *Логика работы команд LDS, LES и LSS.*

Таблица 5.5. Возможные сочетания операндов для команд LDS, LES, LSS.

Приемник	Источник	Пример
R16	Mem32	mem DD ? .... LDS BX, mem

5.1.6. Команды пересылки флагов LAHF и SAHF

Команда **LAHF** (Load AH register from register Flags) копирует в регистр AH содержимое битов CF, PF, AF, ZF, SF регистра FLAGS. А команда SAHF (Store AH register into register Flags), наоборот, восстанавливает эти биты из регистра AH.

Синтаксис:

LAHF
SAHF

Ну, и как вам понравились мнемокоды? Что-то уже проясняется?..

5.2. Арифметические команды

Эта группа команд позволит нам разобраться, как же делает IBM PC простейшие арифметические операции с ЦЕЛОЧИСЛЕННЫМИ данными длиной 8 и 16 бит (частично и с данными длиной 32 бита). Все эти команды сигнализируют о *результате* процесса вычисления, занося определенные значения в соответствующие биты регистра **Flags**.

5.2.1. Команды сложения

Рассмотрим наиболее распространенные команды сложения целочисленных данных. Команды сложения, естественно, складывают числа в двоичной системе счисления. **Команде БЕЗРАЗЛИЧНО знаковые ли это числа или нет.** Если в результате сложения результат НЕ поместился в отведенное место, вырабатывается флаг переноса **CF=1**. В этом случае говорят, что флаг CF *установился*. Из команд сложения команда **ADC** как раз и реагирует на этот флаг — см. п. 5.2.1.3. И вырабатываются еще 4 флага: PF, SF, ZF, OF — см. табл. 5.6.

Таблица 5.6. Состояние флагов после выполнения команд ADD, ADC, INC

Флаги	Пояснение
CF=1	Результат сложения НЕ поместился в операнде-приемнике
PF=1	Результат сложения имеет четное число бит со значением 1
SF=1	Копируется СТАРШИЙ (ЗНАКОВЫЙ) бит результата сложения
ZF=1	Результат сложения равен НУЛЮ
OF=1	Если при сложении двух чисел ОДНОГО знака (ОБА положительные или ОБА отрицательные) результат сложения получился БОЛЬШЕ допустимого значения. В этом случае приемник МЕНЯЕТ ЗНАК.

Из этой таблицы видно, что для чисел со ЗНАКОМ имеют большое значение флаги SF и OF.

5.2.1.1. Команда ADD

Самая простая из этих команд **ADD** (ADDition — сложение).

Синтаксис:

ADD *Приемник*, *Источник*

Логика работы команды:

<*Приемник*> = <*Приемник*> + <*Источник*>

Возможные сочетания операндов для этой команды аналогичны команде **MOV** — см. табл. 5.1.

По сути дела, это — команда сложения с присваиванием, аналогичная принятой в языке C/C++: *Приемник += Источник*;

Недаром у языка C/C++ ноги растут из Ассемблера! И **понимание** ОСНОВ Ассемблера ОЧЕНЬ помогает пониманию языка C/C++.

Видите ли вы, что в результате операции сложения *Приемник портится*, т.е. ИЗМЕНЯЕТ свое значение?

Ну а теперь разберемся подробнее с этой простой командой.

ПРИМЕР 5.4.

Выполнить: **x = a+b**, где **a = 120, b = 100**. Должно получиться **x=220**.

Сразу же возникает вопрос о типе данных. Видно, что и числа, и результат ПОЛОЖИТЕЛЬНЫЕ. Значит, нам достаточно ячейки в 8 бит (один байт) и данные должны быть БЕЗЗНАКОВЫМИ (с точки зрения алгоритмических языков). С точки зрения Ассемблера, как мы знаем (см. п. 2.4.2), главное — это длина операнда — 8 бит. Код, решающий нашу задачу, может иметь следующий вид:

```
; Вариант 1 - код на 2 байта длиннее
mov     aL, 120       ; aL <=== 120
mov     cL, 100       ; cL <=== 100
add     aL, cL        ; <aL>:=<aL>+<cL>
mov     x,  aL        ; x <=== <aL>
; Вариант 2
mov     aL, 120       ; aL <=== 120
add     aL, 100       ; <aL>:=<aL>+100
mov     x,aL          ; x <=== <aL>
```

Для данной постановки задачи, надеюсь, все понятно... Теперь попробуем ее усложнить.

ПРИМЕР 5.4А.

Выполнить: **x = a+b**, где **a**, **b** и **x** — целочисленные данные длиной 8 бит (один байт). В этом случае данные могут быть как знаковые, так и беззнаковые. Для Ассемблера это безразлично (см. модуль AddAs.asm), а вот, например, для Паскаля совсем НЕТ!

Исходный текст модуля AddAs.asm

```
          title addas
; CopyRight by Голубь Н.Г., 1993-1997,2001
          .MODEL Large, Pascal
          ;x=a+b    x,a,b: Byte (ShortInt)
          .data
          Extrn   x:Byte,a:Byte,b:Byte
          .code
          Public  addaS
addaS    proc     far
          mov     aL, a     ; aL <=== a
          mov     cL, b     ; cL <=== b
          add     aL,cL     ; <aL>:=<aL>+<cL>
          mov     x,aL      ; x <=== <aL>
          ret
addaS    endp
          end
```

Сделаем вариант программы на Паскале с вызовом модуля на Ассемблере и проанализируем результаты.

Исходный текст программы AddPb_s.pas

```
Program AddPb_s;
{-----------------------------------------------------------------
                    x = a+b;
              xb=ab + bb;   ab,bb,xb : Byte;
              xs=as + bs;   as,bs,xs : ShortInt;
                  Лекционный пример 5.4a.
            Вариант БЕЗ контроля данных — вар. 1
            CopyRight by Голубь Н.Г., 1993-1997,2001
-----------------------------------------------------------------}
{$I addaSB}    {вызов ASM-модуля addaSB.obj}
{$f+}
Uses CRT; {Может НЕ работать для моделей Celeron и старше... Нужен Path!!!}
var
  ab,bb,xb  : Byte;
  as,bs,xs   : ShortInt;
  ch        : Char;
{Две внешние подпрограммы: для ShortInt и для Byte}
Procedure addaS; {FAR;} external; {FAR НЕ обязателен-есть директива!}
Procedure addaB; external;

begin
  Repeat
   ClrScr;
   Writeln(' Вычислить:      x = a + b; a,b,x : Byte;');
   Write('Введите значение a [0..255] ==== > ');
   Readln(ab);
   Write('Введите значение b [0..255] ==== > ');
   Readln(bb);
   xb := ab+bb;
   Write ('ПАСКАЛЬ:   x = ',xb);
   Writeln (' = ', ab, '+', bb);
   xB:=0;
   addaB;
   write ('АССЕМБЛЕР: x = ',xB);
   Writeln (' = ', ab, '+', bb);
   writeln('==============================================');
   Writeln(' Вычислить:      x = a + b; a,b,x : ShortInt;');
   Write('Введите значение a [-128..127] ==== > ');
   Readln(aS);
   Write('Введите значение b [-128..127] ==== > ');
   Readln(bS);

   xS := aS+bS;
   Write ('ПАСКАЛЬ:   x = ',xS);
   Writeln (' = ', aS, '+', bS);   {Как вы думаете, это для чего???}
   xS:=0;
   addaS;
   write ('АССЕМБЛЕР: x = ',xS);
   Writeln (' = ', aS, '+', bS);
   Writeln('Опять? (y/n)');
   ch:=ReadKey;
   {Readln(ch);}
  Until (ch='n') or (ch='N');
end.
```

Поскольку у нас РАЗНЫЕ типы данных (**ShortInt, Byte**), значит и подпрограмм на Ассемблере должно быть две (хотя они АЛГОРИТМИЧЕСКИ абсолютно одинаковые — разница ТОЛЬКО в *именах данных*). Таким образом, получается немного измененный вариант модуля **AddAs.asm**.

Исходный текст модуля AddaSB.asm

```
            title  addaSB
; CopyRight by Голубь Н.Г., 1993-1997,2001
        .MODEL Large, Pascal
        ;x=a+b    x,a,b:Byte  (ShortInt)
        .data
; Данные описаны в Паскале
        Extrn      xS:Byte,aS:Byte,bS:Byte
        Extrn      xB:Byte,aB:Byte,bB:Byte
        .code
;============  ShortInt  =====================
            Public  addaS
addaS   proc       far
        mov        aL, aS        ; aL <=== a
        mov        cL, bS        ; cL <=== b
        add        aL,cL         ; <aL>:=<aL>+<cL>
        mov        xS,aL         ; x <=== <aL>
        ret
addaS    endp
;==============  Byte  =======================
            Public  addaB
addaB   proc       far
        mov        aL, aB        ; aL <=== a
        mov        cL, bB        ; cL <=== b
        add        aL,cL         ; <aL>:=<aL>+<cL>
        mov        xB,aL         ; x <=== <aL>
        ret
addaB   endp
            end
```

А теперь протестируем наш исполняемый файл **Addpb_s.exe**. Как легко заметить, здесь НЕТ "защиты от дурака". Поэтому ТРИВИАЛЬНЫЕ варианты типа:

Byte: x = 234 = 34 + 200

ShortInt: x = 6 = -44 + 50

будут работать нормально.

Посмотрим, что будет в НЕТРИВИАЛЬНЫХ (*особых*) случаях.

Таблица 5.7. Результаты тестирования (особые случаи)

Byte			**ShortInt**		
Ввод		**Результат на экране дисплея**	**Ввод**		**Результат на экране дисплея**
aB	**bB**	**XB**	**aS**	**bS**	**XS**
435	4	X = 183 = **179** + 4	-423	423	X = 0 = **89** + **-89**
-256	200	X = 200 = **0** + 200	222	2	X = -32 = **-34** + 2
400	-400	X = 0 = **144** + **112**	256	-255	X = 1 = **0** + **1**

Да, результаты интересные... Причем, они одни и те же — и в Паскале, и в Ассемблере. Кстати, и в С/С++ будет то же самое (можете попробовать!). Таким образом получается, что **КОМПИЛЯТОРЫ Turbo/Borland Pascal и Borland С++ НИКАК НЕ РЕАГИРУЮТ НА ФЛАГИ OF, SF И CF!!!** Правда, можно настроить опцию компилятора **Range checking**, тогда мы будем сразу вываливаться из окна MS DOS...

В отладчиках все выдается в шестнадцатеричной системе счисления (что в принципе хорошо и нормально для компьютера, но очень НЕХОРОШО для начинающего программиста). Теперь вы понимаете, что нужно хорошо разбираться в НЕХ-кодах (или хотя бы иметь о них представление).

 ОБРАТИТЕ ВНИМАНИЕ

на выделенные строки в табл. 5.6. Вроде бы РЕЗУЛЬТАТ верный, а данные **a** и **b** восприняты НЕВЕРНО. Это самая неприятная ошибка, потому что ее тяжело поймать, и она может привести к самым неприятным последствиям. **Вот тебе и самая простая операция x = a + b!!!**

5.2.1.2. Обеспечение корректности вычислений на Паскале — вариант 1

Чтобы такого безобразия с исходными данными и результатом НЕ было, сделаем ввод исходных данных на Паскале по возможности достаточно **интеллектуальным**: с ПРОВЕРКОЙ допустимых *вводимых символов*, с КОНТРОЛЕМ вводимого *диапазона чисел*. А также сделаем КОНТРОЛЬ *диапазона для результата*. И если результат при вычислении на Паскале вышел за диапазон, то подпрограмму на Асемблере мы вызывать НЕ будем. Это и есть то, что профессионалы называют "защитой от дурака". Конечно, **программа на Паскале возрастет в размерах (в 3 раза!), но зато станет ОЧЕНЬ надежной**. *А за надежность надо платить.* Модуль **AddaSB.asm** НЕ изменится. Весь удар по корректности данных пока примет на себя Паскаль.

Исходный текст программы AddPb_s.pas

```
Program AddPb_s;
{-----------------------------------------------------------------
                    x = a+b;
              xb=ab + bb;   ab,bb,xb : Byte;
              xs=as + bs;   as,bs,xs : ShortInt;
                   Лекционный пример 5.4a.
            Вариант С КОНТРОЛЕМ данных — вар. 2
             CopyRight by Голубь Н.Г., 1993-1997,2001
-----------------------------------------------------------------}
{$I addaSB}   {вызов ASM-модуля addaSB.obj}
{$f+}
Uses CRT; {Может НЕ работать для моделей Celeron и старше... Нужен Path!!!}
Const
  {Символьные константы}
  inv1='Повторите ввод.';
  inv2='Результат ';
  inv3='Выходит за диапазон [';
  inv4='Вводимое значение ';
  invAb='A (Byte)';
  invBb='B (Byte)';
  invAs='A (ShortInt)';
  invBs='B (ShortInt)';
  {Допустимый диапазон для вводимых данных и результата}
  ShortMin=-128;
```

```
    ShortMax= 127;
    ByteMin= 0;
    ByteMax= 255;
 Label L1;
 var
   ab,bb,xb  : Byte;
   as,bs,xs   : ShortInt;
   ch          : Char;
 Procedure addaS; {FAR;} external; {FAR НЕ обязателен — есть директива!}
 Procedure addaB; external;

 function Fb (a,b:Byte; Var x:Byte) : Boolean;
 Var x1 : Integer;
      s1 : String;
 Begin
   Fb:=True;  {Ошибок НЕТ}
   {ВНИМАНИЕ !!!!!!!!!!! Принципиально важно для корректности вычислений !!!!!!!!
Пояснение см. в примере 5.6}
   x1:=a; {Неявное ПРЕОБРАЗОВАНИЕ Byte ====> Integer}
   x1:=x1+b;
   if (x1>=ByteMin)and(x1<=ByteMax) then x:=x1
    else
      Begin
        s1:=inv2+inv3;  {Формирование сообщения об ошибке}
       Writeln(s1,ByteMin,'..',ByteMax,']!!!!');
       {Вывод полученного значения, выходящего ЗА ДОПУСТИМЫЙ диапазон}
       Writeln(x1);
       Writeln(inv1);
        Fb:=False; {Признак ошибки}
       Exit;
      End
 End;

 function Fs (a,b:ShortInt; Var x:ShortInt) : Boolean;
 Var x1 : Integer;
      s1 : String;
 Begin
   Fs:=True;  {Ошибок НЕТ}
   {ВНИМАНИЕ !!!!!!!!!!! Принципиально важно для корректности вычислений !!!!!!!!}
   x1:=a;
   x1:=x1+b;
   if (x1>=ShortMin)and(x1<=ShortMax) then x:=x1
    else
      Begin
        s1:=inv2+inv3;  {Формирование сообщения об ошибке}
       Writeln(s1,ShortMin,'..',ShortMax,']!!!!');
       {Вывод полученного значения, выходящего ЗА ДОПУСТИМЫЙ диапазон}
       Writeln(x1);
       Writeln(inv1);
        Fs:=False; {Признак ошибки}
       Exit;
      End
 End;
```

```
{Процедура ввода целочисленных данных с проверкой на ДОПУСТИМЫЙ диапазон:
   A               — результат КОРРЕКТНОГО ввода
   inv             — строка приглашения на ввод переменной
   ByteMin,ByteMax — ДОПУСТИМЫЙ диапазон
}
Procedure InputB(Var A:Byte; inv:String; ByteMin,ByteMax:LongInt);
Label L;
Var aL   : Integer;
   s1    : String;
Begin
 L:
  Repeat
    Write('Введите значение ',inv,'===>');
  {Контроль ввода НЕЧИСЛОВЫХ символов}
  {$I-}
    Readln(aL);     {Промежуточный буфер ввода для ЦЕЛОЧИСЛЕННЫХ данных}
  {$I+}
  Until (IOResult=0);
  if (aL>=ByteMin)and(aL<=ByteMax) then  A:=aL
   else
     Begin
       s1:=inv4+inv3;
       Writeln(s1,ByteMin,'..',ByteMax,']!!!!');
       Writeln(inv1);
       goto L;
     End
End;

{Процедура ввода целочисленных данных с проверкой на ДОПУСТИМЫЙ диапазон:
   A               — результат КОРРЕКТНОГО ввода
   inv             — строка приглашения на ввод переменной
   ShortMin,ShortMax — ДОПУСТИМЫЙ диапазон
}
Procedure InputS(Var A:ShortInt; inv:String; ShortMin,ShortMax:LongInt);
Label L;
Var aL   :Integer;
   s1    : String;
Begin
 L:
  Repeat
    Write('Введите значение ',inv,'===>');
  {Контроль ввода НЕЧИСЛОВЫХ символов}
  {$I-}
    Readln(aL);     {Промежуточный буфер ввода для ЦЕЛОЧИСЛЕННЫХ данных}
  {$I+}
  Until (IOResult=0);
  if (aL>=ShortMin)and(aL<=ShortMax) then  A:=trunc(aL)
   else
     Begin
       s1:=inv4+inv3;
       Writeln(s1,ShortMin,'..',ShortMax,']!!!!');
       Writeln(inv1);
       goto L;
     End
End;
```

```
Procedure ExpB;
Label L1;
Begin
  Writeln(' Вычислить:      x = a + b; a,b,x : Byte;');
  L1:
    InputB(aB,invAb,ByteMin,ByteMax);
    InputB(bB,invBb,ByteMin,ByteMax);
    if Fb(aB,bB,xb) then Writeln ('ПАСКАЛЬ:   x=',xB)
      else goto L1; {Результат выходит за диапазон, повторяем ввод данных}
    addaB;
    writeln ('АССЕМБЛЕР: x=',xB);
End;

Procedure ExpS;
Label L1;
Begin
  Writeln(' Вычислить:      x = a + b; a,b,x : ShortInt;');
  L1:
    InputS(aS,invAs,ShortMin,ShortMax);
    InputS(bS,invBs,ShortMin,ShortMax);
    if Fs(aS,bS,xS) then Writeln ('ПАСКАЛЬ:   x=',xS)
      else goto L1; {Результат выходит за диапазон, повторяем ввод данных}
    addaS;
    writeln ('АССЕМБЛЕР: x=',xS);
End;

Procedure Main;
Var i :Integer;
Begin
  Writeln(' Вычислить:      x = a + b;');
  Repeat
    Writeln('Введите Ваш выбор параметров x,a,b');
    Writeln('      1 — Byte');
    Writeln('      2 — ShortInt');
    Writeln('      3 — Выход');
  {Контроль ввода НЕЧИСЛОВЫХ символов}
  {$I-}
    Readln(i);      {Промежуточный буфер ввода для ЦЕЛОЧИСЛЕННЫХ данных}
  {$I+}
  Until (IOResult=0);
  case i of
    1: ExpB;
    2: ExpS;
    3: HALT(0); {Нормальный выход из программы с кодом 0}
  else Exit;
  end
End;

begin
  Repeat
    ClrScr;
    Main;
    Writeln('Повторим? (y/n)');
    ch:=ReadKey;
  Until (ch='n') or (ch='N');
end.
```

Попробуйте такое СРАЗУ сделать на Ассемблере!!! А мы на родном Паскале спокойно отсекаем НЕВЕРНЫЕ комбинации данных (с вежливым сообщением об этом).

Сложение 16-разрядных целых чисел аналогично сложению 8-разрядных чисел, только нужно на Ассемблере для сложения использовать 16-разрядные регистры:

```
; Вариант 1 — код на 2 байта длиннее
    mov     aX, 120        ; aX <=== 120
    mov     cX, 100        ; cX <=== 100
    add     aX, cX         ; <aX>:=<aX>+<cX>
    mov     x,  aX         ; x <=== <aX>
    ; Вариант 2
    mov     aX, 120        ; aX <=== 120
    add     aX, 100        ; <aX>:=<aX>+100
    mov     x,aX           ; x <=== <aX>
```

5.2.1.3. Команда ADC

Эта команда от команды **ADD** отличается использованием бита переноса CF при сложении (ADdition with Carry). Поэтому она может использоваться при сложении 32-разрядных чисел.

Синтаксис:

ADC *Приемник*, *Источник*

Логика работы команды:

<*Приемник*> = <*Приемник*> + <*Источник*> + <CF>

Обычно эта команда работает в паре с командой **ADD** (складывание младших частей числа). Идея работы с младшими и старшими частями числа АБСОЛЮТНО аналогична идее, рассмотренной нами при изучении команды **MOV** — см. пример 5.2:

Фрагмент исходного текста модуля Adda_asm.asm

```
        .MODEL Large, Pascal
        ;x=a+b    x,a,b:LongInt
        .data
        Extrn     x:Dword,a:Dword,b:Dword
        .code
        Public    addaL
addaL   proc      far
        mov       ax,WORD PTR a       ; ax <=== мл. часть a
        mov       bx,WORD PTR a+2     ; bx <=== ст. часть a
        mov       cx,WORD PTR b       ; cx <=== мл. часть b
        mov       dx,WORD PTR b+2     ; dx <=== ст. часть b
        add       ax,cx   ; <ax>:=<ax>+<cx>          мл.часть
        adc       bx,dx   ; <bx>:=<bx>+<dx>+<CF>   ст.часть
        mov       WORD PTR x,ax   ; мл. часть x <=== <ax>
        mov       WORD PTR x+2,bx; ст. часть x <=== <bx>
        ret
addaL   endp
```

5.2.1.4. Обеспечение корректности вычислений на Паскале — вариант 2



ПРИМЕР 5.4В.

Теперь мы можем написать программу сложения 8-, 16- и 32-разрядных чисел. Сделаем программу на Паскале более УНИВЕРСАЛЬНОЙ: создадим модуль, обеспечивающий корректность ввода ЛЮБЫХ ЦЕЛОЧИСЛЕННЫХ исходных данных (Byte, Word, ShortInt, Integer, LongInt).

Исходный текст модуля ввода ЦЕЛЫХ чисел InputNum.pas

```
Unit InputNum;
Interface {Заголовки процедур и функций}
{Процедура ввода целочисленных данных с проверкой на ДОПУСТИМЫЙ диапазон:
   A            — результат КОРРЕКТНОГО ввода
   inv          — строка приглашения на ввод переменной
   Min,Max      — ДОПУСТИМЫЙ диапазон
}
Const
   {Символьные константы}
  inv1='Повторите ввод.';
  inv3='выходит за диапазон [';
  inv4='Вводимое значение ';
Procedure InputNumber(Var A:LongInt; inv:String; Min,Max:LongInt);

Implementation {Реализация процедур и функций}

Procedure InputNumber(Var A:LongInt; inv:String; Min,Max:LongInt);
Label L;
Var aL   :Real;  {!!!!!!!!!!!!!! Внимание !!!!!!!!!!!!!!!!!!}
    s1      : String;
Begin
  L:
  Repeat
    Write('Введите значение ',inv,'===>');
  {Контроль ввода НЕЧИСЛОВЫХ символов}
  {$I-}
    Readln(aL);     {Промежуточный буфер ввода для ЦЕЛОЧИСЛЕННЫХ данных}
  {$I+}
  Until (IOResult=0);
  if (aL>=Min)and(aL<=Max) then  A:=trunc(aL)
   else
     Begin
      s1:=inv4+inv3;
      Writeln(s1,Min,'..',Max,']!!!!');
      Writeln(inv1);
      goto L;
     End
End;
End.
```

 **ОБРАТИТЕ ВНИМАНИЕ.**

Чтобы обеспечить ПРОВЕРКУ исходных данных на ДОПУСТИМЫЙ диапазон, ВВОДИМОЕ данное должно быть описано, как минимум, на тип выше. Т.е. для контроля диапазона 8-разрядных чисел вводимое данное должно быть описано, как минимум, 16-разрядное — см. п. 5.2.1.2. В нашем случае мы заявили, что могут быть данные и LongINT. Поэтому и появляется такое описание:

```
Var aL    :Real;
```

Исходный текст ГЛАВНОЙ программы сложения любых ЦЕЛЫХ чисел AddBSIWL.pas

```
Program AddBSIWL;
{-----------------------------------------------------------------
                    x=a+b;
                 a,b,x : Integer;
                 a,b,x : ShortInt;
                 a,b,x : LongInt;
                 a,b,x : Byte;
                 a,b,x : Word;
                Лекционный пример 5.4b.
              Вариант С КОНТРОЛЕМ данных
          CopyRight by Голубь Н.Г., 1993-1997,2001
-----------------------------------------------------------------}
{$I adda_asm}    {вызов ASM-модуля adda_asm.obj}
{$f+}
Uses CRT, InputNum;
Const
  {Символьные константы}
  inv2='Результат ';
  invAi='A (Integer)';
  invBi='B (Integer)';
  invAs='A (ShortInt)';
  invBs='B (ShortInt)';
  invAb='A (Byte)';
  invBb='B (Byte)';
  invAw='A (Word)';
  invBw='B (Word)';
  invAl='A (LongInt)';
  invBl='B (LongInt)';
  {Допустимый диапазон для вводимых данных и результата}
  IntMin=-32768;
  IntMax= 32767;
  WordMin=    0;
  WordMax= 65535;
  ShortMin=-128;
  ShortMax= 127;
  ByteMin=  0;
  ByteMax= 255;
  LongMin: LongInt=-2147483647-1;
  LongMax= 2147483647;
Label L1;
var  {Описание ГЛОБАЛЬНЫХ переменных}
  a,b,x          : LongInt;
  xS,aS,bS     : ShortInt;
  xB,aB,bB     : Byte;
```

```
   xl,al,bl        : Integer;
   xW,aW,bW : Word;
   ch              : Char;

 {Внешние процедуры на Ассемблере}
 Procedure addal(var x:Integer; a,b:Integer); external;
 Procedure addaS(var x:ShortInt; a,b:ShortInt); external;
 Procedure addaL(var x:LongInt; a,b:LongInt);  external;
 Procedure addaW(var x:Word; a,b:Word); external;
 Procedure addaB(var x:Byte; a,b:Byte);  external;

 {Функция вычисления x=a+b с проверкой РЕЗУЛЬТАТА на допустимый диапазон}
 function F (a,b:LongInt; Var x:LongInt; Min,Max:Longint) : Boolean;
 Var x1 : Real;
     s1   : String;
 Begin
   F:=True;  {Ошибок НЕТ}
   {ВНИМАНИЕ !!!!!!!!!!! Принципиально важно для корректности вычислений !!!!!!!!
                                                  Пояснение см. в примере 5.6}
   x1:=a;
   x1:=x1+b;
   if (x1>=Min)and(x1<=Max) then x:=trunc(x1)
    else
      Begin
        s1:=inv2+inv3;  {Формирование сообщения об ошибке}
       Writeln(s1,Min,'..',Max,']!!!!');
        {Вывод полученного значения, выходящего ЗА ДОПУСТИМЫЙ диапазон}
       Writeln(x1);
       Writeln(inv1);
        F:=False; {Признак ошибки}
       Exit;
      End
 End;

 {—— Блок процедур работы с ОПРЕДЕЛЕННЫМ типом данных ——}
 Procedure Expl;  {Integer}
 Label L1;
 Begin
  Writeln(' Вычислить:      x = a + b; a,b,x : Integer;');
  L1:
    InputNumber(a,invAi,IntMin,IntMax);
    InputNumber(b,invBi,IntMin,IntMax);
    if F(a,b,x,IntMin,IntMax) then Writeln ('ПАСКАЛЬ:   x=',x)
      else goto L1; {Результат выходит за диапазон, повторяем ввод данных}
    al:=a;bl:=b;
    addal(xl,al,bl);
    writeln ('АССЕМБЛЕР: x=',xl);
 End;

 Procedure ExpW;  {Word}
 Label L1;
 Begin
  Writeln(' Вычислить:      x = a + b; a,b,x : Word;');
  L1:
    InputNumber(a,invAw,WordMin,WordMax);
    InputNumber(b,invBw,WordMin,WordMax);
    if F(a,b,x,WordMin,WordMax) then Writeln ('ПАСКАЛЬ:   x=',x)
```

```
    else goto L1; {Результат выходит за диапазон, повторяем ввод данных}
   aW:=a;bW:=b;
   addaW(xW,aW,bW);
   writeln ('АССЕМБЛЕР: x=',xW);
End;

Procedure ExpB; {Byte}
Label L1;
Begin
 Writeln(' Вычислить:      x = a + b; a,b,x : Byte;');
 L1:
   InputNumber(a,invAb,ByteMin,ByteMax);
   InputNumber(b,invBb,ByteMin,ByteMax);
   if F(a,b,x,ByteMin,ByteMax) then Writeln ('ПАСКАЛЬ:   x=',x)
    else goto L1; {Результат выходит за диапазон, повторяем ввод данных}
   aB:=a;bB:=b;
   addaB(xB,aB,bB);
   writeln ('АССЕМБЛЕР: x=',xB);
End;

Procedure ExpS;{ShortInt}
Label L1;
Begin
 Writeln(' Вычислить:      x = a + b; a,b,x : ShortInt;');
 L1:
   InputNumber(a,invAs,ShortMin,ShortMax);
   InputNumber(b,invBs,ShortMin,ShortMax);
   if F(a,b,x,ShortMin,ShortMax) then Writeln ('ПАСКАЛЬ:   x=',x)
    else goto L1; {Результат выходит за диапазон, повторяем ввод данных}
   aS:=a;bS:=b;
   addaS(xS,aS,bS);
   writeln ('АССЕМБЛЕР: x=',xS);
End;

Procedure ExpL; {LongInt}
Label L1;
Begin
 Writeln(' Вычислить:      x = a + b; a,b,x : LongInt;');
 L1:
   InputNumber(a,invAl,LongMin,LongMax);
   InputNumber(b,invBl,LongMin,LongMax);
   if F(a,b,x,LongMin,LongMax) then Writeln ('ПАСКАЛЬ:   x=',x)
    else goto L1; {Результат выходит за диапазон, повторяем ввод данных}
   x:=0;
   addaL(x,a,b);
   writeln ('АССЕМБЛЕР: x=',x);
End;
{—————————————————————————————————————————}
Procedure Main;
Var i :Integer;
Begin
 Writeln(' Вычислить:      x = a + b;');
 Repeat
   Writeln('Введите Ваш выбор параметров x,a,b');
   Writeln('     1 — Integer');
   Writeln('     2 — Word');
```

```
    Writeln('     3 — Byte');
    Writeln('     4 — ShortInt');
    Writeln('     5 — LongInt');
    Writeln('     6 — Выход');
  {Контроль ввода НЕЧИСЛОВЫХ символов}
  {$I-}
    Readln(i);     {Промежуточный буфер ввода для ЦЕЛОЧИСЛЕННЫХ данных}
  {$I+}
  Until (IOResult=0);
  case i of
    1: ExpI;
    2: ExpW;
    3: ExpB;
    4: ExpS;
    5: ExpL;
    6: HALT(0); {Нормальный выход из программы с кодом 0}
  else Exit;
  end
End;

begin
  Repeat
    ClrScr;
    Main;
    Writeln('Повторим? (y/n)');
    ch:=ReadKey;
  Until (ch='n') or (ch='N');
end.
```

Исходный текст модуля Adda_asm.asm

```
          title Adda_asm.asm
; CopyRight by Голубь Н.Г., 1993-1997, 2001
          .MODEL Large, Pascal
          ;x=a+b
; x,a,b:LongInt, Byte, ShortInt, Integer, Word
          .data
          Extrn    x:Dword,a:Dword,b:Dword
          Extrn    xS:Byte,aS:Byte,bS:Byte
          Extrn    xB:Byte,aB:Byte,bB:Byte
          Extrn    xI:Word,aI:Word,bI:Word
          Extrn    xW:Word,aW:Word,bW:Word
          .code
          Public   addaL
addaL     proc     far
          mov      ax,WORD PTR a        ; ax <=== мл. часть a
          mov      bx,WORD PTR a+2 ; bx <=== ст. часть a
          mov      cx,WORD PTR b        ; cx <=== мл. часть b
          mov      dx,WORD PTR b+2 ; dx <=== ст. часть b
          add      ax,cx   ; <ax>:=<ax>+<cx>          мл.часть
          adc      bx,dx   ; <bx>:=<bx>+<dx>+<CF>   ст.часть
          mov      WORD PTR x,ax        ; мл. часть x <=== <ax>
          mov      WORD PTR x+2,bx ; ст. часть x <=== <bx>
          ret
addaL     endp
          Public   addaS
```

```
addaS    proc     far
          mov      aL, aS      ; aL <=== aS
          mov      cL, bS      ; cL <=== bS
          add      aL,cL       ; <aL>:=<aL>+<cL>
          mov      xS,aL       ; xS <=== <aL>
          ret
addaS    endp
          Public   addaB
addaB    proc     far
          mov      aL, aB      ; aL <=== aB
          mov      cL, bB      ; cL <=== bB
          add      aL,cL       ; <aL>:=<aL>+<cL>
          mov      xB,aL       ; xB <=== <aL>
          ret
addaB    endp
          Public   addaI
addaI    proc     far
          mov      ax, aI      ; ax <=== aI
          mov      cx, bI      ; cx <=== bI
          add      ax,cx       ; <ax>:=<ax>+<cx>
          mov      xI,ax       ; xI <=== <ax>
          ret
addaI    endp
          Public   addaW
addaW    proc     far
          mov      ax, aW      ; ax <=== aW
          mov      cx, bW      ; cx <=== bW
          add      ax,cx       ; <ax>:=<ax>+<cx>
          mov      xW,ax       ; xW <=== <ax>
          ret
addaW    endp
          end
```

Эти модули и программа будут **базовыми** для ВСЕХ наших дальнейших программ.

5.2.1.5. Команда INC

Мнемокод этой команды получен в результате сокращения такого предложения: INCrement operand by 1 — Увеличение значения операнда на 1. Команда эта содержит один **операнд** и имеет следующий **синтаксис**:

INC *Операнд*

Логика работы команды:

< *Операнд* > = < *Операнд* > + 1

В качестве операнда допустимы регистры и память: R8, R16, Mem8, Mem16. Например,

INC I ; I++
INC ax ; <AX> = <AX> + 1
INC cL ; <CL> = <CL> + 1

5.2.2. Команды вычитания SUB, SBB, DEC, NEG и CMP

Команды вычитания **SUB**, **SBB**, **DEC** ОБРАТНЫ соответствующим командам сложения **ADD**, **ADC** и **INC**. Они имеют те же самые операнды.

SUB (SUBtract — Вычитание).

SBB (SuBtract with Borrow CF — Вычитание с заемом флага переноса CF).

DEC (DECrement operand by 1 — Уменьшение значения операнда на 1).

ПРИМЕР 5.5.

Сделаем вычитание 32-разрядных чисел. Оно принципиально НИЧЕМ отличаться от сложения НЕ будет. Воспользуемся для решения нашей задачи имеющимися заготовками из файла **Adda_asm.asm**.

Исходный текст модуля Subal.asm

```
   title subaL
         ;x=a-b
data     segment para public
          Extrn    x:Dword,a:Dword,b:Dword
data     Ends
code     segment para public
   assume  cs:code,ds:data
   Public  suba
suba     proc     far
          mov       ax,WORD PTR a
          mov       bx,WORD PTR a+2
          mov       cx,WORD PTR b
          mov       dx,WORD PTR b+2
          sub       ax,cx
          sbb       bx,dx
          mov       WORD PTR x,ax
          mov       WORD PTR x+2,bx
          ret
suba     endp
code         ends
          end
```

В данной реализации модуля мы применили директивы описания сегментов SEGMENT, ENDS и ASSUME. **Обратите внимание** на ИМЕНА СЕГМЕНТОВ, которые мы стыкуем с Паскалем — см. п. 4.3.

Команда **NEG** (NEGate operang — изменение знака операнда).

Синтаксис:

NEG *Операнд*

Логика работы команды:

< *Операнд* > = — < *Операнд* >

В качестве операнда допустимы регистры и память: R8, R16, Mem8, Mem16. Например, для вычисления <AL> = 50 — <AL> следующий код:

```
NEG AL
ADD AL,50
```

будет более эффективен, чем такой:

```
MOV  BL, 50
SUB  BL, AL
MOV  AL,BL
```

Это связано с тем, что операция ВЫЧИТАНИЕ более МЕДЛЕННАЯ, чем операция СЛОЖЕНИЕ.

Команда **CMP** (CoMPare operands — сравнение операндов) аналогична команде **SUB**, однако в отличие от нее ПРИЕМНИК НЕ изменяется, но зато получаются флаги (которые затем анализируются командами УСЛОВНОГО ПЕРЕХОДА — см. п. 9.2).

Таблица 5.8. Состояние флагов после выполнения команд SUB, SBB, DEC, NEG, CMP.

Флаги	Пояснение
CF=1	Требуется заем разряда при вычитании
PF=1	Результат вычитания имеет четное число бит со значением 1
SF=1	Копируется СТАРШИЙ (ЗНАКОВЫЙ) бит результата вычитания
ZF=1	Результат вычитания равен НУЛЮ
OF=1	Если при вычитании двух чисел с РАЗНЫМИ знаками результат вычитания получился БОЛЬШЕ допустимого значения. В этом случае приемник МЕНЯЕТ ЗНАК.

5.2.3. Команды умножения MUL и IMUL

А вот это действительно одни из САМЫХ НЕПРИЯТНЫХ команд целочисленной арифметики для базовой конфигурации IBM PC XT из-за наличия *неявных* операндов. MUL (MULtiply — БЕЗЗНАКОВОЕ умножение) и IMUL (Integer MULtiply — ЗНАКОВОЕ целочисленное умножение). Как видно из определения команд, они учитывают наличие ЗНАКА.

Синтаксис:

MUL *Источник*
IMUL *Источник*

Логика работы команд:

*<Произведение> = <Множимое> * <**Источник_Множитель**>*

Итак, *Источником* является *Множитель*. В качестве *Множителя* допустимы регистры и память: R8, R16, Mem8, Mem16. **Константы НЕ допускаются!!!** А где же находятся *Множимое* и *Произведение*? Вот это и есть источник ошибок — НЕЯВНЫЕ операнды. Они находятся в СТРОГО ОПРЕДЕЛЕННОМ МЕСТЕ в зависимости от **длины** *Источника*.

Таблица 5.9. Неявные операнды команд MUL, IMUL

Длина источника (Множителя)	Множимое	Произведение
Byte	AL	AX (<AH:AL>)
Word	AX	<DX:AX>

Итак, сделаем следующие **выводы**:

- Команда IMUL реагирует на ЗНАК перемножаемых чисел.
- Расположение *Множимого* и *Произведения* — **строго определенное** и зависит од ДЛИНЫ *Множителя* — это нужно просто ЗАПОМНИТЬ.
- Длина *Произведения* всегда в **ДВА** раза больше, чем у *Множителя*. Причем старшая часть *Произведения* находится либо в регистре AH, либо в DX. **Именно об ЭТОМ обстоятельстве ЗАБЫВАЮТ как люди, так и компиляторы** (их ведь тоже делали люди!) — см. пример 5.6.

Эти команды тоже вырабатывают флаги — см. табл. 5.6. Но устанавливаются флаги **CF=1** и **OF=1**, если результат слишком велик для отведенных ему регистров назначения.



ПРИМЕР 5.6.

Вычислить произведение для БЕЗЗНАКОВЫХ 16-разрядных данных **z:=5*w**; где **w:WORD; z:LongInt;** (с точки зрения **Turbo/Borland Pascal** — у него НЕТ БЕЗЗНАКОВОГО 32-разрядного представления, поэтому принимаем, что результат должен быть **z:LongInt**).

Исходный текст модуля MULword.asm

```
          title MULword
          ;z=5*w
data     segment para public
          Extrn    w:word,z:Dword
data     Ends
code     segment para public
   assume  cs:code,ds:data
   Public  MULword
MULword proc     far
          mov      cx,5     ; Множитель 5 ===> CX
          mov      AX,w     ; Множимое w ====> AX
          MUL      cx        ; <DX:AX> = <AX>*<CX>
          mov      WORD PTR z,ax    ; Младшая часть
          mov      WORD PTR z+2,DX ; Старшая часть
          ret
MULword endp
code         ends
          end
```

Исходный текст программы MULwordp.pas

```
program MULwordp;
{---------------------------------------------------------------------
                    z:=5*w;
                  w:WORD; z:LongInt;
             Лекционный пример 5.6 — вариант 1.
               Вариант БЕЗ контроля данных
           CopyRight by Голубь Н.Г., 1993-1997,2001
---------------------------------------------------------------------}
{$I MULword}
{$f+}
USES CRT;
```

```
var w    :word;{!!!!!! данное БЕЗ знака !!!!!!!!!}
    z    :LongInt; {Другого НИЧЕГО 32-разрядного на Паскале НЕТ}
    ch   :char;
Procedure MULword; external;
Procedure MULp;
Begin
  z:=5*w; { Считает ВЕРНО, если z = [0..65535] !!!!!!!!!!!!!!!!!! }
End;
Begin
  ch:='y';
  while (ch='y') or (ch='Y') do
    begin
      write ('введите w [0..65535]:  ');
      readln (w);
      MULword;
      writeln ('АССЕМБЛЕР:  z=',z);
      MULp;
      writeln ('ПАСКАЛЬ:  z=',z);
      writeln ('продолжать? (y/n)');
      ch:=ReadKey;
    end
End.
```

Если мы протестируем этот пример, то убедимся, что **ПАСКАЛЬ (чудеса!!!) НЕ всегда считает ВЕРНО**, а вот Ассемблер считает НОРМАЛЬНО:

W = 222
Z = 1110 (И Паскаль, и Ассемблер)
W = 11111
Z = 55555 (И Паскаль, и Ассемблер)
W = 22222
Z = 111110 (Ассемблер)
Z = **45574** (Паскаль ??????????)
W = 44444
Z = 222220 (Ассемблер)
Z = **25612** (Паскаль ??????????)

 **ПОЯСНЕНИЕ**

В чем же тут дело? Оказывается, при ЛЮБЫХ арифметических вычислениях (в Паскале, да и в C/C++ тоже!), если производятся операции с ОДИНАКОВЫМИ младшими типами данных, результат сначала получается ТАКОГО же типа, а при присваивании делается ПРИВЕДЕНИЕ младших типов данных к старшим. В нашем случае мы имеем константу 5 (по умолчанию она 16-разрядная) и 16-разрядную переменную w. **Поэтому результат ПРОИЗВЕДЕНИЯ получается тоже 16-разрядным**. В то же время при вычислениях с РАЗНЫМИ типами данных делается ПРИВЕДЕНИЕ младших типов данных к старшим ДО вычисления. Поэтому, чтобы получить результат 32-разрядным, **нужно один из операндов** сделать тоже 32-разрядным (например, так: **z:=5;**). Теперь мы можем умножать: **z:=z*w;** и получим ПРАВИЛЬНЫЙ результат.

Фрагмент исправленного исходного текста программы MULwordp.pas

```
Procedure MULp;
Begin
  z:=5; {!!!!!!! Преобразование 5 ===> LongInt !!!!!!!!}
  z:=z*w;  {Теперь ВСЕ нормально!}
End;
```

Умножение для 8-разрядных (знаковых и беззнаковых данных), а также для 16-разрядных знаковых по идее АБСОЛЮТНО аналогично. Можете для тренировки выполнить их сами — это ОЧЕНЬ полезно и покажет, действительно ли вы все поняли...

А вот с умножением 32-разрядных чисел мы познакомимся позже — см. главу 12.

5.2.4. Команды деления DIV и IDIV

А это — САМЫЕ НЕПРИЯТНЫЕ команды целочисленной арифметики. Они источник 95-100% ошибок для начинающих осваивать Ассемблер. Корень зла, конечно, коренится в командах **MUL** и **IMUL**, поскольку из математики известно, что умножение и деление — взаимно обратные операции. Но чтобы понимать, что же происходит при делении, надо хотя бы чуть-чуть иметь понятие о внутреннем представлении целочисленных данных — см. п. 2.2.

DIV (DIVide — БЕЗЗНАКОВОЕ деление), **IDIV** (Integer DIVide — ЗНАКОВОЕ деление целых чисел).

Синтаксис:

DIV *Источник*
IDIV *Источник*

Логика работы команд:

<*Частное:Остаток*> = <*Делимое*> / <***Источник_Делитель***>

Итак, *Источником* является *Делитель*. В качестве *Делителя* допустимы регистры и память: R8, R16, Mem8, Mem16. **Константы НЕ допускаются!!!** А где же находятся *Делимое* и <*Частное:Остаток*>? Опять источник ошибок — НЕЯВНЫЕ операнды. Они находятся тоже в СТРОГО ОПРЕДЕЛЕННОМ МЕСТЕ в зависимости от **длины** *Делителя*.

Таблица 5.10. Неявные операнды команд DIV, IDIV.

Длина источника (Делителя)	Делимое	Результат	
		Частное	Остаток
Byte	AX (<AH:AL>)	AL	AH
Word	<DX:AX>	AX	DX

Выводы:

- Команда **IDIV** реагирует на ЗНАК обрабатываемых чисел.
- Расположение *Делимого* и *Результата* — **строго определенное** и завиоит од ДЛИНЫ *Делителя* — это нужно просто ЗАПОМНИТЬ.
- *Результат* состоит из *Частного* и *Остатка*, которым при обычных целочисленных вычислениях пренебрегают.

- Длина *Делимого* всегда в **ДВА** раза больше, чем у *Делителя*. Причем старшая часть *Делимого* находится либо в регистре АН, либо в DX. **Именно об ЭТОМ обстоятельстве тоже ЗАБЫВАЮТ как люди, так и компиляторы** — см. пример 5.7 (вариант 1).

Эти команды тоже вырабатывают флаги — см. табл. 5.6. Но устанавливаются флаги **CF=1** и **OF=1**, если частное НЕ помещается в регистры AL или АХ. Здесь, в отличие от команд умножения, **всегда генерируется ПРЕРЫВАНИЕ "Деление на ноль"**, хотя на ноль мы и НЕ думаем делить — см. пример 5.7.

ПРИМЕР 5.7.

Вычислить **z:=5*w/4;** где **w,z:WORD**. Итак, продолжаем исследовать БЕЗЗНАКОВЫЕ умножение и деление. Воспользуемся предыдущим примером. В результате умножения **5*w** у нас сформировались два регистра <DX:AX>, поэтому без особых проблем (в Ассемблере) мы можем сделать деление на 4 (нуля, как видно, здесь и близко НЕТ!!!)

Исходный текст модуля Divword.asm

```
            title divword
            ;z=5*w/4
data       segment para public
            Extrn    w:word,z:word
data       Ends
code       segment para public
   assume   cs:code,ds:data
   Public   divword
divword proc      far
            mov      cx,5      ; Множитель 5 ===> CX
            mov      AX,w      ; Множимое w ====> AX
            MUL      cx        ; <DX:AX> = <AX>*<CX>
            mov      cx,4
            div      cx                  ; 5*w/4
            mov      z,ax
            ret
divword endp
code           ends
            end
```

Исходный текст программы DivwordP.pas

```
program divwordp;
{-------------------------------------------------------------
                    z:=5*w/4;
                   w,z:WORD;
            Лекционный пример 5.7 — вариант 1.
              Вариант БЕЗ контроля данных
          CopyRight by Голубь Н.Г., 1993-1997,2001
-------------------------------------------------------------}
{$I divword}
{$f+}
USES CRT;
var w,z   :word;{!!!!!! данные БЕЗ знака !!!!!!!!}
   ch    :char;
Procedure divword; external;
Procedure divp;
```

```
Begin
{Могут быть ситуации "ДЕЛЕНИЕ на НОЛЬ"!!!!!!!!!!!!
        или НЕВЕРНЫЙ результат — см. тесты}
  z:=5*w div 4
End;

Begin
  ch:='y';
  while (ch='y') or (ch='Y') do
    begin
      write ('введите w [0..65535]:  ');
      readln (w);
      divword;
      writeln ('АССЕМБЛЕР:  z=',z);
      divp;
      writeln ('ПАСКАЛЬ:  z=',z);
      writeln ('продолжать? (y/n)');
      ch:=ReadKey;
    end
End.
```

Теперь протестируем нашу задачу:

```
введите w [0..65535]:   11111
АССЕМБЛЕР:   z=13888
ПАСКАЛЬ:   z=13888
продолжать? (y/n)
введите w [0..65535]:   22222
АССЕМБЛЕР:   z=27777
ПАСКАЛЬ:   z=11393
продолжать? (y/n)
введите w [0..65535]:   33333
АССЕМБЛЕР:   z=41666
ПАСКАЛЬ:   z=8898
продолжать? (y/n)
введите w [0..65535]:   44444
АССЕМБЛЕР:   z=55555
ПАСКАЛЬ:   z=6403
продолжать? (y/n)
введите w [0..65535]:   55555
!!!!! Деление на НОЛЬ !!! (Runtime Error 200 at ...)
```

Паскаль опять врет... И в конце концов мы выходим на абсурдную в данном случае ситуацию — деление на ноль! Вспомним наши рассуждения по поводу примера 5.6 и применим их в данном случае.

Исправленный фрагмент исходного текста программы DivwordP.pas

```
function divp:Boolean;
Var temp : LongInt;
Begin
  divP:=True;
  temp := 5;{!!!!!!!!!!!!!! Внимание !!!!!!!!!!!!!!!!!!}
{Теперь НЕ могут быть ситуации "ДЕЛЕНИЕ на НОЛЬ"!!!!!!!!!!!!!}
  temp:=temp*w div 4;
  if temp >= 65535 then
    begin
      Writeln('Полученный результат ', temp,
            ' превышает допустимый диапазон [0..65535]!!!');
```

```
      Writeln ('Можете попробовать ввести ДРУГИЕ данные...');
      divP:=False
    end;
  z:=temp;
End;
```

Теперь протестируем нашу задачу еще раз:

```
введите w [0..65535]:   11111
АССЕМБЛЕР:   z=13888
ПАСКАЛЬ:          z=13888
продолжать? (y/n)
введите w [0..65535]:   22222
АССЕМБЛЕР:   z=27777
ПАСКАЛЬ:        z=27777
продолжать? (y/n)
введите w [0..65535]:   33333
АССЕМБЛЕР:   z=41666
ПАСКАЛЬ:        z=41666
продолжать? (y/n)
введите w [0..65535]:   44444
АССЕМБЛЕР:   z=55555
ПАСКАЛЬ:        z=55555
продолжать? (y/n)
введите w [0..65535]:   55555
Полученный результат 69443 превышает допустимый диапазон
   [0..65535]!!!
Можете попробовать ввести ДРУГИЕ данные...
продолжать? (y/n)
```

Теперь все нормально. Вот так знания Ассемблера помогают нам корректно решить достаточно простую задачу на Паскале.

5.2.5. Команды распространения знака CBW и CWD

Начнем сначала с примера.



ПРИМЕР 5.8.

Вычислить **z=w/10** для **всевозможных сочетаний** 8- и 16-разрядных типов целочисленных данных.

Исходный текст модуля DivA.asm — вар. 1

```
            title divA
; CopyRight by Голубь Н.Г., 2001
;             Корректно считается НЕ всегда:
; НЕТ формирования регистров DX (AH) ПЕРЕД делением
;!!! Возможны ситуации ДЕЛЕНИЕ на НОЛЬ!!!
          ;z=w/10
data     segment para public
          Extrn    wW:word,zW:word ; Word
          Extrn    wI:word,zI:word ; Integer
          Extrn    wB:BYTE,zB:BYTE ; Byte
          Extrn    wS:BYTE,zS:BYTE ; ShortInt
data     Ends
```

```
code     segment para public
   assume   cs:code,ds:data
   Public   divA
divA     proc      far
;============= WORD =================================
           mov       AX,wW     ; Множимое w ====> AX
           mov       cx,10
           div       cx        ; <AX>=<DX:AX>/<CX>=w/10
           mov       zW,ax
;
           mov       AX,wI     ; Множимое w ====> AX
           Idiv      cx        ; <AX>=<DX:AX>/<CX>=w/10
           mov       zI,ax
;============= Byte =================================
           mov       AL,wB     ; Множимое w ====> AL
           mov       cL,10
           div       cL        ; <AL>=<AX>/<CL>= w/10
           mov       zB,aL
;
           mov       AL,wS     ; Множимое w ====> AL
           Idiv      cL        ; <AL>=<AX>/<CL>= w/10
           mov       zS,aL
           ret
divA     endp
code         ends
           end
```

Если запустим этот модуль совместно с главным модулем **DivP.pas** на счет, мы получим достаточно странные результаты со стороны Ассемблера. Основная причина этого в том, что **мы совершенно НЕ позаботились о подготовке ДЕЛИМОГО к делению**, которое, как известно (см. табл. 5.10), должно быть в ДВА раза ДЛИННЕЕ делителя. Это и есть САМАЯ ГЛАВНАЯ наша ОШИБКА.

Как же подготовить делимое? В примере 5.7 мы об этом НЕ заботились — за нас это сделала команда умножения. В примере 5.8 ее нет. Поэтому будем это делать САМИ. Здесь уместно вспомнить о внутреннем представлении целых чисел — см. п. 2.2.

БЕЗЗНАКОВЫЕ числа занимают всю ячейку памяти, понятие знак для них НЕ существует — они считаются ПОЛОЖИТЕЛЬНЫМИ. Поэтому при делении БЕЗЗНАКОВЫХ чисел **в СТАРШУЮ часть делимого надо занести НОЛЬ**. Это можно сделать уже известными нам командами: MOV AH,0 или MOV DX,0. Однако это *НЕ эффективно*, поэтому, забегая немного вперед, мы используем родную для компьютера *команду сложения по модулю 2* (см. п. 7.1.3): **XOR AH,AH** или **XOR DX,DX**.

Для **ЗНАКОВЫХ данных** существуют две замечательные команды распространения знака. **CBW** (Convert Byte toWord — преобразовать байт, находящийся в регистре AL, в слово — регистр AX) и **CWD** (Convert Word to Double word — преобразовать слово, находящееся в регистре AX, в двойное слово — регистры <DX:AX>). Операнды им НЕ нужны.

Синтаксис:

CBW

CWD

Эти команды также применяются при выравнивании операндов по длине.

Таблица 5.11. Логика работы команды CBW при получении знакового делимого.

1) Получаем старшую часть (AH)	Известна младшая часть (AL)	
Биты 15-8	Знак - бит 7	Биты 6-0
1111 1111	1	Информационная часть числа
0000 0000	0	Информационная часть числа
2) Получаем ЗНАКОВОЕ делимое - регистр AX (AH:AL)		

Таблица 5.12. Логика работы команды CWD при получении знакового делимого.

1) Получаем старшую часть (DX)	Известна младшая часть (AX)	
Биты 31-16	Знак - бит 15	Биты 14-0
1111 1111 1111 1111	1	Информационная часть числа
0000 0000 0000 0000	0	Информационная часть числа
2) Получаем ЗНАКОВОЕ делимое - регистры <DX:AX>		

Если мы захотим распространить знак для БЕЗЗНАКОВЫХ данных (вслушайтесь, как ФАЛЬШИВО звучит эта часть фразы...), мы и получим в большинстве случаев "**Деление на ноль**".

Если вы разобрались во внутреннем представлении целочисленных данных, значит, НИКОГДА не ошибетесь в командах **DIV** и **IDIV**. Внесем в наш пример необходимые коррективы.

Исправленный исходный текст модуля DivA.asm — вар. 2

```
         title divA
; CopyRight by Голубь Н.Г., 2001
         ;z=w/10
data     segment para public
         Extrn    wW:word,zW:word ; Word
         Extrn    wI:word,zI:word ; Integer
         Extrn    wB:BYTE,zB:BYTE ; Byte
         Extrn    wS:BYTE,zS:BYTE ; ShortInt
data     Ends
code     segment para public
   assume  cs:code,ds:data
   Public  divA
divA     proc     far
```

```
;============= WORD ================================
            mov     AX,wW   ; Множимое w ====> AX
            mov     cx,10
            XOR     DX,DX   ; !!!!!!!!!!! <DX>=0 !!!!!!!
            div     cx      ; <AX>=<DX:AX>/<CX>=w/10
            mov     zW,ax
;
            mov     AX,wI   ; Множимое w ====> AX
            CWD             ; !!! Распространение знака AX
            Idiv    cx      ; <AX>=<DX:AX>/<CX>=w/10
            mov     zI,ax
;============= Byte ================================
            mov     AL,wB   ; Множимое w ====> AL
            mov     cL,10
            XOR     AH,AH   ; !!!!!!!!!!! <AH>=0 !!!!!!!
            div     cL      ; <AL>=<AX>/<CL>= w/10
            mov     zB,aL
;
            mov     AL,wS   ; Множимое w ====> AL
            CBW             ; !!! Распространение знака AL
            Idiv    cL      ; <AL>=<AX>/<CL>= w/10
            mov     zS,aL
            ret
divA      endp
code            ends
            end
```

Исходный текст программы DivP.pas

```
program divPascal;
{------------------------------------------------------------
                    z:=w/10;
          Лекционный пример 5.8 — вариант 1,2.
            Вариант БЕЗ контроля данных
        CopyRight by Голубь Н.Г., 1993-1997,2001
------------------------------------------------------------}
{$I divA}
{$f+}
USES CRT;
var  wW,zW  :word;
     wI,zI  :Integer;
     wB,zB  :Byte;
     wS,zS  :ShortInt;
     ch     :char;
Procedure divA; external;
Procedure divp;
Begin
  zW:=wW div 10;
  zI:=wI div 10;
  zB:=wB div 10;
  zS:=wS div 10;
End;
```

```
Begin
  ch:='y';
  while (ch='y') or (ch='Y') do
    begin
      writeln ('=========         z:=w/10;         ==============');
      write ('введите w [0..65535]:  ');
      readln (wW);
      write ('введите w [-32768..32767]:  ');
      readln (wI);
      write ('введите w [0..255]:  ');
      readln (wB);
      write ('введите w [-128..127]:  ');
      readln (wS);
      divA;
      writeln ('———— АССЕМБЛЕР:—————');
      writeln ('WORD      при ', wW, ' z=',zW);
      writeln ('Integer   при ', wI, ' z=',zI);
      writeln ('BYTE      при ', wB, ' z=',zB);
      writeln ('ShortInt  при ', wS, ' z=',zS);
      divp;
      writeln ('———— ПАСКАЛЬ:—————');
      writeln ('WORD      при ', wW, ' z=',zW);
      writeln ('Integer   при ', wI, ' z=',zI);
      writeln ('BYTE      при ', wB, ' z=',zB);
      writeln ('ShortInt  при ', wS, ' z=',zS);
      writeln ('продолжать? (y/n)');
      ch:=ReadKey;
    end
End.
```

Теперь с точки зрения Ассемблера все нормально! А с точки зрения Паскаля? Попробуйте протестировать задачу и САМОСТОЯТЕЛЬНО устранить неполадки. Вам это уже по силам!



Ассемблер и языки высокого уровня

У всякого портного свой взгляд на искусство!

Козьма Прутков

Если ничто другое не помогает, прочтите, наконец, инструкцию!

Аксиома Кана
(Артур Блох "Закон Мерфи")

Известно, что язык Ассемблера предоставляет программисту полную свободу действий при разработке программ. Но он требует известной программистской культуры и знания архитектуры компьютера. Использование алгоритмических языков (языков высокого уровня) существенно снижает требования к программисту и позволяет очень быстро (особенно при склонности к алгоритмическому мышлению) научиться писать сначала несложные, а потом и достаточно сложные программы.

Но в жизни каждого **серьезного программиста**, пишущего на одном из алгоритмических языков, однажды наступает момент, когда он НЕ может понять, почему компьютер, например, при одних данных считает верно, а при других — выдает невесть что. Мы с вами этот серьезный момент в жизни уже опытного программиста спровоцировали в самом начале изучения языка Ассемблера — см. пример 5.4а. Обычно начинающие программисты с большим удивлением смотрят на такого рода провокации — ведь они же изучали данный алгоритмический язык и всегда все вроде бы было нормально. Вот поэтому и существуют бета-тестеры, знающие, как правило, не один алгоритмический язык и, конечно, язык Ассемблера, которые видят узкие места в программе.

Мы с вами уже тоже кое-что увидели. Например, что нельзя втиснуть в ячейку информации больше, чем она может вместить. Мы уже знаем, как хранятся данные в памяти компьютера, и немного знаем, как компьютер делает такие простые и естественные для нас с вами арифметические операции. Теперь пришла пора эти наши знания обобщить и пойти дальше в изучении Ассемблера.

Сосредоточимся на описании соглашений по стыковке разноязычных модулей, принятых в среде **Borland Pascal** и **Borland C/C++**. Эта среда, на взгляд автора, более всего приспособлена для начинающих осваивать премудрости программирования в операционных системах линейки Windows. Набравшись немного опыта, займемся анализом разработок и фирмы Microsoft (Visual C++, MASM) — внимательный читатель, возможно, уже заметил, что мы с самого начала НЕ игнорировали эту фирму.

6.1. Соглашения по интерфейсу

Когда программист пишет программу, он должен соблюдать определенные внешние правила:

- Выдерживать синтаксис операторов языка, иначе компилятор не поймет, что же мы хотим сделать.
- Придерживаться определенной структуры оформления своих модулей/программ.

Когда же стыкуются два разноязычных модуля, эти требования по структуре каждого из модулей (*интерфейсу*) становятся очень важными, иначе данные (или подпрограммы) из одного модуля станут недоступны командам другого модуля. Обычно в этом случае свое возмущение высказывает компоновщик: "Не могу что-то (*конкретно что*) найти". Мы этих вопросов уже немного касались в п. 4.8.1 и в примерах главы 5. Теперь мы, вооруженные уже некоторым опытом, вернемся к ним.

6.1.1. Borland/Turbo Pascal — 7.0x и Turbo Assembler

6.1.1.1. Основные требования к модулю на языке Паскаль

Перечислим ряд **важнейших** требований, взяв за основу пример 5.1. Затем, по мере освоения материала, мы эти требования (равно, как и примеры) будем расширять и уточнять. Но принцип стыковки Паскаль+Ассемблер нам уже будет понятен.

1. Указать, какой конкретно модуль на языке Ассемблера (ASM-модуль) вас интересует — в нашем случае **Mov_16.asm**.

2. Этот модуль должен быть уже оттранслирован соответствующим компилятором Ассемблера и в результате получен объектный файл — **Mov_16.obj**. Лучше всего брать родной компилятор **tasm.exe** версии **3.2**, который входит в стандартную поставку Borland/Turbo Pascal. Мы уже знаем, что в случае ошибок объектный файл НЕ выдается. Поиск ошибок приятнее искать по файлу листинга — **Mov_16.lst**. Компиляцию **asm**-модуля можно делать прямо в интегрированной среде, набрав **Shift-F3** (обычно так по умолчанию вызывается компилятор **tasm.exe** версии **3.2**).

3. Использовать специальную директиву Паскаля **{$L** *имя***}** для загрузки (Load) этого объектного модуля — получим **{$L Mov_16.obj }**.

4. Описать внешние переменные как ГЛОБАЛЬНЫЕ (иначе компоновщик Паскаля и, соответственно, модуль на Ассемблере при вычислениях их НЕ увидят). В нашем случае примера 5.1 получаем следующее описание:

```
Var
   x,a    : Integer;
   b,z    : Byte;
```

5. Описать внешние (external) процедуры или функции, реализованные на Ассемблере, в синтаксисе языка Паскаль. Для нашего примера это будет описание процедуры:
```
Procedure Mov_16(var x:integer; var z:Byte); external;
```
6. Если задача достаточно сложная, нужно решить ее и на Паскале, для чего описать соответствующие процедуры и функции (см. пример 5.4b).
7. Сделать ввод исходных данных. Конечно, лучше сделать ввод исходных данных с "защитой от дурака" — см. пример 5.4a (п. 5.2.1.2) или пример 5.4b (п.5.2.1.4).
8. В нужном месте вызвать наши процедуры или функции. В примере 5.1 мы вызвали единственную, внешнюю функцию, реализованную на языке Ассемблера:
```
mov_16(x,z);
```
9. Выдать полученные результаты и сравнить их.

ЗАМЕЧАНИЕ.

У среды программирования **Borland/Turbo Pascal 7.0x** есть очень интересная особенность. Внешнюю ассемблерную процедуру вы можете описать на Паскале и вовсе **БЕЗ параметров**, например, так:
```
Procedure Mov_16; external;
```
И, соответственно, вызвать ее следующим образом:
```
mov_16;
```
Или **перечислить ВСЕ параметры**:
```
Procedure Mov_16(a:Integer; b:Byte;
              var x:integer; var z:Byte); external;
.............................................
mov_16(a,b,x,z);
```
Модуль на Ассемблере от этого НЕ изменится. Главное, за чем надо следить — ВНЕШНИЕ имена должны быть ГЛОБАЛЬНЫМИ и того же самого типа.

Внимательный читатель спросит, а почему ничего не было сказано о директиве Паскаля *дальнего* (**FAR**) *вызова* ВНЕШНИХ процедур или функций? Наличие этой директивы определяется тем, какая *модель памяти* используется в модуле на Ассемблере. В нашем примере (кстати, и во всех остальных тоже) используется модель **LARGE** (как более устойчивая в операционных системах линейки Windows, особенно в 32-разрядных Windows NT 4 или Windows 2000, на которых и проверялись все примеры). Синтаксис этой директивы допускает два толкования:

- {$F+}
- *заголовок_внешней_процедуры/функции*; FAR; External;

В данном случае вы можете применять в своих программах на Паскале любую из них (и даже обе вместе!). Главное, чтобы какая-то из них была. Иначе рискуете зависнуть по памяти при запуске окна **MS DOS** из **Microsoft Windows** (особенно из **Windows NT4/2000**).

6.1.1.2. Основные требования к модулю на языке Ассемблера

Внешние данные, получаемые из языка Паскаль, должны *обязательно* храниться в *сегменте данных* и описываться с помощью директивы **EXTRN**. Сами процедуры или функции, естественно, должны храниться в сегменте кода и иметь атрибут **PUBLIC**. Язык стыковки (**Pascal**) для современных версий компиляторов (**tasm 3.x** или выше) можно НЕ писать — *он принят по умолчанию.* Приведем два варианта интерфейса для ASM-модуля, используя реальные примеры 5.1 и 5.5.

```
              TITLE Вариант 1 ———————-
        .MODEL Large
;====== Описание ВНЕШНИХ (ГЛОБАЛЬНЫХ) данных
        .data
        x,a:Integer
        Extrn    a: WORD, x:word
        ;z,b:byte
        Extrn    b: BYTE, z:BYTE
;====== Описание ВНЕШНИХ процедур или функций
        .code
        Public   Mov_16
Mov_16   proc      far
        ; ———————— команды
        ret
Mov_16   endp
        Public   Name
Name    proc      far
        ; ———————— команды
        ret
Name    endp
.............................................
        end

              TITLE Вариант 2 ———————-
;====== Описание ВНЕШНИХ (ГЛОБАЛЬНЫХ) данных
DATA    SEGMENT
        ;x,a:Integer
        Extrn    a: WORD, x:word
        ;z,b:byte
        Extrn    b: BYTE, z:BYTE
DATA    EndS
;====== Описание ВНЕШНИХ процедур или функций
CODE    SEGMENT
        assume   cs:code,ds:data
        Public   Mov_16
Mov_16  proc      far
        ; ———————— команды
        ret
Mov_16  endp
        Public   Name
Name    proc      far
        ; ———————— команды
        ret
Name    endp
.............................................
CODE    EndS
        end
```



ОБРАТИТЕ ВНИМАНИЕ.

Имена сегментов данных и кода должны быть СТРОГО ОПРЕДЕЛЕННЫМИ — см. п. 4.3.

6.1.2. Borland C++ и Turbo Assembler

Стыковка модулей Ассемблера с модулями, написанными на языке C/C++, более хитрая (равно, как и сам алгоритмический язык!). Нужно просто быть БОЛЕЕ внимательным и ТОЧНО следовать инструкциям.

C и C++ являются **различными** языками, хотя между ними есть много общего. C — это процедурный язык, а C++ — объектно-ориентированный. Они продолжают развиваться, каждый в свою сторону. Среда программирования **Borland C++** имеет в своем составе *комбинированный* компилятор, который реагирует *на расширение файлов*: ***.c** — модуль на языке **C**, ***.cpp** — модуль на языке **C++**. Кроме того, фирма **Borland** славится тем, что дает программисту возможность *управлять процессом компиляции* с помощью множества дополнительных настроек. Конечно, для начинающего программиста это все может быть ОЧЕНЬ непонятно, поэтому *мы ограничимся настройками, принятыми по умолчанию.* Кроме того, компиляторы, естественно, тоже развиваются вместе с языками, поэтому то, что прекрасно поддерживает одна версия, не всегда, к сожалению, адекватно поддерживает другая. Поэтому в конкретных примерах всегда будет указываться версия среды программирования или компилятора C/C++.



ПРИМЕР 6.1.

Реализуем отдельно на языке C и C++ (в интегрированной среде программирования **Borland C++ 5.02**) вычисление арифметического выражения **x/3** для целочисленного значения **x** (знакового и беззнакового, платформа **Win16 — EasyWin**). Пока корректность вводимых данных проверять НЕ будем.

Исходный текст программы на языке C (c.c)

```
/* c.c     Borland C++ 5.02        EasyWin
   (c) Copyright 2000 by Голубь Н.Г.
   ВНИМАНИЕ!!! Диапазон вводимых данных НЕ проверяется !!!!!!!!!!!!
*/
#include <stdio>
// функции компонуются как функции языка C.
int funcCi(int x)
{ return x/3;}
int funcCu(unsigned int x)
{ return x/3;}
void main()
{int a,t=0;
  unsigned b;
  for (;;)
   { printf ("\n============ test #%d ===================",++t);
     printf ("\n???? a (int) =======> ");
     scanf  ("%d", &a);
     printf ("\nЯзык C: %d/3 = %d",a, funcCi(a));
     printf ("\n???? b (unsigned)=======> ");
     scanf  ("%u", &b);
     printf ("\nЯзык C: %u/3 = %u",b, funcCu(b));
   }
 }
```

Если мы запустим эту программу на счет, получим следующее:

```
============ test #1 ===================
???? a (int) =======> 11111
Язык С: 11111/3 = 3703
???? b (unsigned)=======> 44444
Язык С: 44444/3 = 14814
============ test #2 ===================
???? a (int) =======> 22222
Язык С: 22222/3 = 7407
???? b (unsigned)=======> 3333
Язык С: 3333/3 = 1111
============ test #3 ===================
???? a (int) =======> -222222
Язык С: -25614/3 = -8538
???? b (unsigned)=======> -111111
Язык С: 19961/3 = 6653
```



ОБРАТИТЕ ВНИМАНИЕ НА ТЕСТ #3.

За корректностью данных компилятор здесь тоже по умолчанию НЕ следит, как и в Паскале.

Изменим в этой программе расширение и назовем полученный файл **c1.cpp**. Теперь наша программа *компилятором* будет рассматриваться как программа на языке С++. Внешне это почти НЕ будет заметно (**c1.exe** станет немного короче, чем **c.exe**), но зато **внутренние** изменения будут. И для нас именно эти внутренние изменения — ОЧЕНЬ существенные.

6.1.2.1. Искажение имен в языке С++

Посмотрим же на эти внутренние изменения. Для этого попросим компилятор командной строки **bcc.exe** (он входит в стандартную поставку ВС++ 5.02 и имеет версию 5.2) выдать нам файл на языке Ассемблера (Produce assembly output):

```
bcc -S C1.cpp
bcc -S C.c
```



ВНИМАНИЕ.

Обратите внимание на ключ **-S** (Source). Буква **S** должна быть ЗАГЛАВНАЯ!!!

Если среда программирования ВС++ установлена НЕ корректно, компилятор НЕ сможет найти месторасположение библиотеки. В этом случае с помощью ключа —I*путь* нужно явно указать путь к библиотеке подключаемых файлов, например, так:

```
bcc -Id:\BC5\INCLUDE -S C.c
```

В результате мы получим файлы **c.asm** и **c1.asm** — это и есть результат работы компилятора С/С++ по *переводу кода С/С++ в код Ассемблера.*

Рассмотрим, чем же эти файлы отличаются. По умолчанию в код добавляется ОТЛАДОЧНАЯ информация (в частности, ОЧЕНЬ подробные комментарии), которая поможет нам посмотреть, как транслируются операторы языка С/С++. От параметров мы пока отвлечемся. Они ПЕРЕД вызовом функций заносятся в стек (подробнее см. п. 9.1.2.4), а затем оттуда извлекаются и обрабатываются.

Оказывается, что эти файлы отличаются своими ВНУТРЕННИМИ ИМЕНАМИ функций! Происходит это потому, что, в отличие от языка C, функции языка C++ — это ОЧЕНЬ гибкая конструкция, которая может *перегружаться* в зависимости от количества и типа параметров. Поэтому **формальные параметры специально кодируются во ВНУТРЕННИХ именах функций C++. В языке C (BC++) к имени функции, которую дал ей программист, просто добавляется знак подчеркивания**. Кстати, этот знак добавляется и ко всем ВНУТРЕННИМ и ВНЕШНИМ именам переменных, массивов, структур и проч., НЕЗАВИСИМО от того, на каком языке написана программа (на языке C или на языке C++). Именно на этот знак подчеркивания и реагирует компоновщик в первую очередь. **Поэтому БЕЗ ОСОБОЙ НУЖДЫ программисту НЕ рекомендуется начинать СВОИ имена со знака подчеркивания — пусть это останется привилегией компилятора!**

Фрагмент текста модуля c.asm (получен при компиляции файла c.c)

```
_TEXT   segment byte public "CODE"
  assume    cs:_TEXT,ds:DGROUP
_funcCi proc near
   ;
   ;   int funcCi(int x)
   ;
  push bp
  mov  bp,sp
   ;
   ;   {  return x/3;}
   ;
  mov  ax,word ptr [bp+4]
  mov  bx,3
  cwd
  idiv bx
  pop  bp
  ret
_funcCi endp
  assume    cs:_TEXT,ds:DGROUP
_funcCu proc near
   ;
   ;   int funcCu(unsigned int x)
   ;
  push bp
  mov  bp,sp
   ;
   ;   {  return x/3;}
   ;
  mov  ax,word ptr [bp+4]
  mov  bx,3
  xor  dx,dx
  div  bx
  pop  bp
  ret
_funcCu endp
```

Фрагмент текста модуля c1.asm (получен при компиляции файла c1.cpp)

```
_TEXT   segment byte public "CODE"
   assume    cs:_TEXT,ds:DGROUP
@funcCi$qi   proc near
    ;
    ;   int funcCi(int x)
    ;
   push bp
   mov  bp,sp
    ;
    ;   {  return x/3;}
    ;
   mov  ax,word ptr [bp+4]
   mov  bx,3
   cwd
   idiv bx
   pop  bp
   ret
@funcCi$qi   endp
   assume    cs:_TEXT,ds:DGROUP
@funcCu$qui  proc near
    ;
    ;   int funcCu(unsigned int x)
    ;
   push bp
   mov  bp,sp
    ;
    ;   {  return x/3;}
    ;
   mov  ax,word ptr [bp+4]
   mov  bx,3
   xor  dx,dx
   div  bx
   pop  bp
   ret
@funcCu$qui  endp
```

Как видите, **разница в именах функций весьма существенная!** А сам код на Ассемблере в основном вам уже должен быть понятен.

6.1.2.2. Спецификатор сборки extern

Можно сделать функции C++ совместимыми с функциями языка C, используя спецификатор сборки (Linkage specifer). Он сообщает компилятору, что одна или более функций программы на C++ будет компоноваться как функция языка C:

extern "C" { *прототипы_ функций;* **}**

Все спецификаторы функций должны быть глобальными, а сами имена функций — уникальными. В нашем случае это будет выглядеть следующим образом:

Исходный текст программы c2.cpp

```
/* c2.cpp  Borland     C++ 5.02     EasyWin
   (c) Copyright 2000 by Голубь Н.Г.
   Спецификатор сборки extern.
   ВНИМАНИЕ!!! Диапазон вводимых данных !!!!!!!!!!!!!!!
*/
```

```
#include <stdio>
//!!!!!!!!!!!!!! имена функций должны быть УНИКАЛЬНЫМИ !!!!!!!!!!
extern "C"          // компонуется как функция С
{   int funcCi(int x);
    int funcCu(unsigned int x);
 }
int funcCi(int x)
{ return x/3;}
int funcCu(unsigned int x)
{ return x/3;}
void main()
{ int a,t=0;
    unsigned b;
    for (;;)
  { printf ("\n============= test #%d ===================",++t);
          printf ("\n???? a (int) =======> ");
          scanf ("%d", &a);
          printf ("\nЯзык C++: %d/3 = %d",a, funcCi(a));
          printf ("\n???? b (unsigned)=======> ");
          scanf ("%u", &b);
          printf ("\nЯзык C++: %u/3 = %u",b, funcCu(b));
  }
}
```

Теперь, если сделать файл **c2.asm**, можно убедиться, что имена наших функций в этом файле стали такими же, как и в файле **c.asm**.

Этот прием позволяет БЕЗ проблем компоновать и ассемблерные программы с программами на С++.

6.1.2.3. Основные требования к главному модулю на языке С++ (ВС++ 4.5, 5.02)

1. Имена внешних переменных и функций должны быть описаны, как глобальные.
2. Имена внешних функций должны быть уникальными.
3. К именам внешних функций должен быть применен спецификатор сборки

   extern "C" { *прототипы_ функций*; },

   чтобы они интерпретировались в стиле языка С.
4. Константы ВНЕШНИМИ быть НЕ могут.
5. Вызов внешних функций делается, как обычно это принято в языке С++.
6. **Стыкуемые модули должны быть описаны в проекте**. Поскольку мы сейчас изучаем 16-разрядное программирование на Ассемблере, проект должен быть ТОЛЬКО **Application[.exe] | DOS (standard)**.

Рассмотрим на конкретном примере вопросы стыковки модулей С++ и Ассемблер для выполнения программ С++ в окне MS DOS для линейки операционных систем Windows.

ПРИМЕР 6.2.

Вычислить на Ассемблере и на С++ следующее арифметическое выражение:

X = (2*a+b*c)/(d-a); int a, b = -333, c = 1000, d = -10 , X;

Предусмотреть *проверку диапазона* для вводимого значения переменной **a** и результата **X**. Значения **b**, **c** и **d** считать ЛОКАЛЬНЫМИ константами. Сравнить значения полученных результатов.

Будем решать нашу задачу по шагам (среда BC++ 5.02).

1. Реализуем сначала поставленную задачу в знакомом нам языке C++. Получим, например, следующую программу:

```
/* PrimC.cpp               Borland C++ 5.02
           Проект !!!!! Application.exe === > DOS (standard) !!!!!
                        asm+cpp файлы
          (c) Copyright 1993-1998-2000 by Голубь Н.Г.
            Вычислить  x=(2*a+b*c)/(d-a);  int a,b,c,d,x;
*/

#include <conio.h>
#include <iostream.h>
#include <limits.h>

inline int  test(long int a)    // (c) Copyright 1998 by студент Артем Астафьев
    {return ((a>>15)+1)&~1;}

int     primC(int a, const int b, const int c, const int d)
{ double z = (2.0*a+1.0*b*c)/(d-a);  //!!!!!!!!!! ВНИМАНИЕ !!!!!!!!!!!!!!!!
  if (z > SHRT_MIN && z < SHRT_MAX) return z;
  else
  { cout << "\n!!!!!!!!!!! Limits of int value !!!!!!!!!!!\n  x = "
        << z << endl;
   return SHRT_MIN; //-32768 !!!!!!!!!!!!
  }
}

void main(void)
{
   // ЛОКАЛЬНЫЕ константы
   const b=-333;
   const c= 1000;
   const d=-10;
   // ЛОКАЛЬНОЕ описание переменных
   long int a1;
   int X,a;
   char ch;
   do {X = 0;
      //clrscr();
      cout << "\n    x=(2*a+b*c)/(d-a);  int x,a,b=-333,c=1000,d=-10;" << endl;
      do{ cout<<"\n    Enter a [-32768..32767], a!= " << d << " ====> ";
         cin>> a1;  //!!!!!!!!!! ВНИМАНИЕ !!!!!!!!!!!!!!!!
         }while (test(a1) || d — a1 == 0 || test (d-a1));
      a = a1;
      X = primC(a, b, c, d);
      if (X!=SHRT_MIN)
       { cout << " Result (C++) x = " << X << endl;
       }
      cout << "\n\nExit? — (y/n)\n";
      ch = getch();
     } while (!(ch=='y' || ch =='Y'));
}
```

 **ОБРАТИТЕ ВНИМАНИЕ!**

Как изящно, со знанием дела (а именно: машинного представления знаковых целочисленных данных) студент второго курса Артем Астафьев реализовал функцию **int test(long int a)** для анализа допустимого диапазона [-32768..32767].

Проверим наши вычисления:

```
        x=(2*a+b*c)/(d-a);    int  x,a,b=-333,c=1000,d=-10;
        Enter  a  [-32768..32767],  a!=  -10  ====>  22
   Result  (C++)  x  =  10404
Exit?  -  (y/n)
        x=(2*a+b*c)/(d-a);    int  x,a,b=-333,c=1000,d=-10;
        Enter  a  [-32768..32767],  a!=  -10  ====>  -10
        Enter  a  [-32768..32767],  a!=  -10  ====>  11111111111111
        Enter  a  [-32768..32767],  a!=  -10  ====>  1
   Result  (C++)  x  =  30272
Exit?  -  (y/n)
        x=(2*a+b*c)/(d-a);    int  x,a,b=-333,c=1000,d=-10;
        Enter  a  [-32768..32767],  a!=  -10  ====>  -1
!!!!!!!!!!!  Limits  of  int  value  !!!!!!!!!!!
   x  =  37000.2
```

Задача, реализованная на языке C++, работает нормально. НЕ сделана только проверка на корректность ввода допустимых символов. Но такая задача перед нами пока и НЕ ставилась.

2. Опишем ВНЕШНЮЮ ГЛОБАЛЬНУЮ ассемблерную функцию **prim** (пока БЕЗ параметров) в стиле языка С:

```
extern "C"
{    void prim (void);}
```

3. Перенесем ЛОКАЛЬНОЕ в функции **main** описание ВНЕШНИХ переменных **int X, a;** на ГЛОБАЛЬНЫЙ уровень.

4. Если результат функции, вычисленной в C++, НЕ превышает допустимый диапазон, сделаем вызов ассемблерной функции и вывод полученного результата:

```
X = 0;
prim();
cout << " Result (ASM) x = " << X << endl;
```

Окончательный вариант исходного текста программы PrimC.cpp

```
/* PrimC.cpp              Borland C++ 5.02
   Проект !!!!!! Application.exe === > DOS (standard) !!!!!!
                          asm+cpp файлы
          (c) Copyright 1993-1998-2000 by Голубь Н.Г.
             Вычислить  x=(2*a+b*c)/(d-a);  int a,b,c,d,x;
*/

#include <conio.h>
#include <iostream.h>
#include <limits.h>

inline  int    test(long int a) // (c) Copyright 1998 by студент Артем Астафьев
   {return ((a>>15)+1)&~1;}
```

```
int     primC(int a, const int b, const int c, const int d)
{ double z = (2.0*a+1.0*b*c)/(d-a); //!!!!!! ВНИМАНИЕ !!!!!
  if (z > SHRT_MIN && z < SHRT_MAX) return z;
   else
   { cout << "\n!!!!!!! Limits of int value !!!!!!!!\n x ="
         << z << endl;
     return SHRT_MIN; //-32768
   }
}
extern "C"
{ void prim (void);}
int X,a;
void main(void)
{ char ch;
    const b=-333;
    const c= 1000;
    const d=-10;
    long int a1;
    do {X = 0;
            //clrscr();
            cout << "\n x=(2*a+b*c)/(d-a); int x,a,b=-333,c=1000,d=-10;" << endl;
            do{ cout<<"\n Enter a [-32768..32767], a!=" <<d<< "====>";
                    cin>> a1; //!!!!!!!!!! ВНИМАНИЕ !!!!!!!!!!!!!!!
                    }while (test(a1) || d — a1 == 0 || test (d-a1));
            a = a1;
            X = primC(a, b, c, d);
            if (X!=SHRT_MIN)
            { cout << " Result (C++) x = " << X << endl;
              X = 0; // для чистоты эксперимента !!!
              prim();
              cout << " Result (ASM) x = " << X << endl;
            }
            cout << "\n\nExit? — (y/n)\n";
            ch = getch();
            } while (!(ch=='y' || ch =='Y'));
}
```

Теперь займемся модулем на языке Ассемблера.

6.1.2.4. Основные требования к модулю на языке Ассемблера

1. Модель желательно делать **LARGE** (во избежание проблем с оперативной памятью, особенно в 32- и 64-разрядных операционных системах) с ОБЯЗАТЕЛЬНЫМ указанием языка программирования **C**:

   ```
   model   large,C
   ```

2. **Внешние переменные**, передаваемые из CPP-модуля, **могут быть описаны как в сегменте данных, так и в сегменте кода** с помощью директивы **EXTRN**.
3. **Описание констант должно быть ПОВТОРЕНО**. Они должны быть ЛОКАЛЬНЫМИ и могут быть расположены как в сегменте данных, так и в сегменте кода.

4. Внешние функции должны быть описаны с помощью директивы описания общих имен с ОБЯЗАТЕЛЬНЫМ указанием языка программирования С:

PUBLIC C *имя_внешней_функции.*

5. При стыковке модулей в среде ВС++ описание сегментов лучше делать в родном стиле **Ideal: CODESEG** и **DATASEG** (хотя можно и в стиле MASM: **.Code** и **.Data**).

Исходный текст модуля Prim.asm

```
            title Арифметические выражения
            model   large,C
            ; tasm PRIM.ASM /l /ml !!!!!!!!!!!!!!!!!!!!!!!!
            ; CopyRight by Голубь Н.Г., 1993-1997, 2000
            ; Пример 6.2.
            ;x=(2*a + b*c)/(d-a), d<>a !!!
            ;int x,a,b,c,d;
            CODESEG
;====== ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
        Extrn    C X:word
        Extrn    C a:word
; ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ — КОНСТАНТЫ
b       dw   -333
c       dw   1000
d       dw   -10
;=================================================
        Public  C prim
prim    proc far
        mov  ax,2
        Imul a     ; <dx>:<ax>=2*a
        mov  bx,dx ; bx <=== ст.часть (dx)
        mov  cx,ax ; cx <=== мл.часть (ax)
        mov  ax,b
        Imul c     ; <dx>:<ax>=b*c
        add  ax,cx ; <ax>=<ax>+<cx>   (мл.часть)
        adc  dx,bx ; <dx>=<dx>+<bx>   (ст.часть)
        mov  cx,d
        sub  cx,a  ; <cx>=<cx>-a
        Idiv cx    ; <ax>=<dx>:<ax>/<cx>
        mov  X,ax
        ret
prim    endp
        end
```

6.1.2.5. Основные шаги по созданию проекта DOS (standard) для ВС++ 4.x (5.x)

1. **Имена файлов и папок** должны быть, как в DOS, **<= 8 символов**. Назовем наши файлы **PrimC.cpp** и **Prim.asm**. Создадим их здесь же, в редакторе ВС++ 5.02.

2. Компилятор **tasm.exe** желателен версии 4.1 (3.2). Мы будем использовать компилятор **tasm.exe** версии 4.1, который входит в стандартную поставку TASM-5. Путь к нему системе должен быть известен. Особенно это касается операционных систем Windows NT 4 и Windows-2000: путь (**Path**) в переменных среды должен быть указан ВРУЧНУЮ и, чтобы это сработало, система должна перезагрузиться.

3. Создадим проект. Он должен быть только **Application[.exe] | DOS (standard)** — см. рис.6.1. Назовем наш проект **primc**.

4. Теперь наполним наш проект конкретным содержанием. А именно: какие конкретно файлы мы хотим стыковать — см. рис. 6.2.

5. Теперь можем запустить все на компиляцию и счет. "Умный" компилятор BC++ 4.x (5.x) все сделает сам: и откомпилирует разноязычные модули, и соберет их, и запустит полученный **exe**-файл на счет (ЕСЛИ ВЫ ВСЕ ШАГИ СДЕЛАЛИ ПРАВИЛЬНО!).

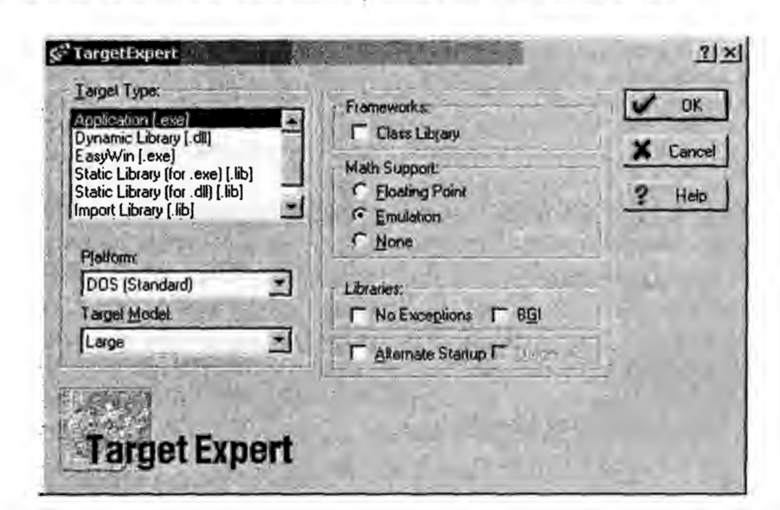


РИС. 6.1. *Проект для стыковки cpp- и asm-файлов для компиляторов BC++ 4.x (5.x).*

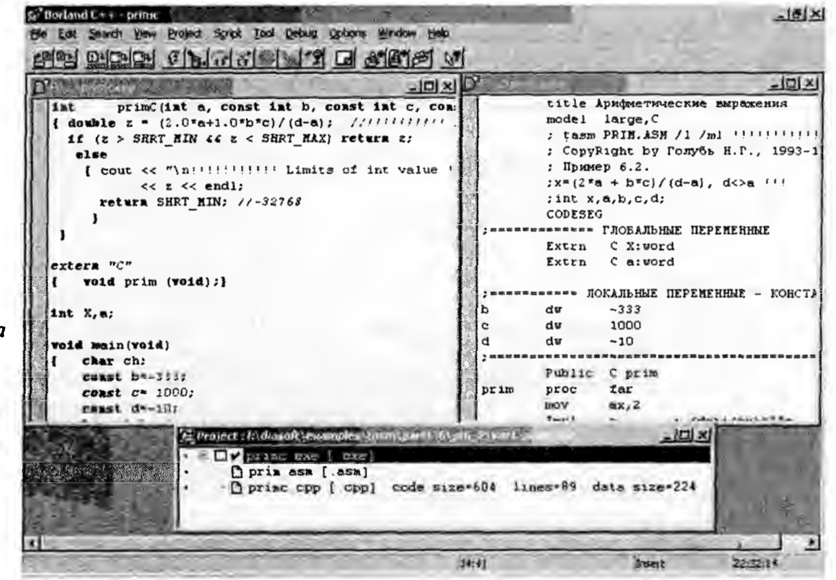


РИС. 6.2. *Вид экрана исходных файлов* ***(PrimC.cpp*** *и* ***Prim.asm)*** *и файла проекта* ***primc.ide*** *в интегрированной среде программирования* ***Borland C++5.02***



ЗАМЕЧАНИЕ.

Если компилятор **tasm.exe** в процессе создания **exe**-файла НЕ будет найден (это значит, что вы *плохо* дружите со своей операционной системой — см. п. 2 требований), можно **asm**-файл откомпилировать ОТДЕЛЬНО. Мы это уже не один раз делали, но теперь нужно это сделать с УЧЕТОМ РЕГИСТРА ВНЕШНИХ символов (добавляется ключ **/ml** — см. п. 4.8.1):

tasm PRIM.ASM /l /ml

Если НЕТ ошибок, мы получим **obj**-файл (**Prim.obj**), который нужно ВМЕСТО **ASM**-файла включить в проект (см. п. 4). Это, конечно, лишняя морока, здесь ОЧЕНЬ легко ошибиться. Но за все надо платить: или подружитесь со своей операционной системой, или перейдите на Паскаль+Ассемблер (см. п. 6.1.1), или займитесь классикой (просмотр результатов через отладчик — см. п. 5.1.1.3), или программируйте пока на встроенном Ассемблере, или ... попробуйте воспользоваться компилятором командной строки **bcc.exe**. Возможно, вы помните, что он нам уже помогал — см. п. 6.1.2.1. И вообще компилятор командной строки — хорошая вещь!

Для НОРМАЛЬНОЙ работы с компилятором командной строки сделаем достаточно интеллектуальный командный (**bat**) файл, в котором пропишем все пути и вызовы. **Bat**-файл также сделает нам **lst**-файл **asm**-файла и контроль правильности введенных параметров. На моем компьютере, например, пути и вызовы выглядят так, как указано в файле **Make.bat**. Наверняка у вас НЕ так. Надо просто этот файл соответствующим образом отредактировать. И у вас ВСЕ должно ПОЛУЧИТЬСЯ!

Командный файл Make.bat

```
@echo off
if -%1 == - goto NoParam
if -%2 == - goto NoParam
if not exist %1 goto NotExist1
if not exist %2 goto NotExist2
goto Ok
:NotExist1
echo ERROR!!!
type %1
echo Not Exist!!!
goto exit
:NotExist2
echo ERROR!!!
type %2
echo Not Exist!!!
goto exit
:NoParam
echo ERROR!!!
echo Variant of a call:
echo make.bat labasm_4.cpp laba4.asm
goto exit
:Ok
set path=%path%;N:\bc5\bin;N:\bc5\lib;N:\TASM5\BIN
set include=N:\BC5\INCLUDE
set lib=N:\BC5\LIB
set link=N:\BC5\BIN
tasm %2 temp.obj /l /ml
bcc -ml %1 temp.obj
if exist *.obj del *.obj >nul
:exit
```

Теперь мы можем запустить наш командный файл:

MAKE.BAT PrimC.cpp PRIM.ASM

и БЕЗ особых проблем получим файлы **Prim.lst** и **Primc.exe** (последний файл даже меньшей длины).

Протокол наших действий:

```
F:\DiaSoft\Examples\Tasm\Part1\6\Pr6_2\Var5>MAKE.BAT PrimC.cpp PRIM.ASM
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996 Borland International

Assembling file:      PRIM.ASM  to  temp.obj
Error messages:       None
Warning messages:     None
Passes:               1
Remaining memory:     436k

Borland C++ 5.2 Copyright (c) 1987, 1997 Borland International
Mar 19 1997 17:29:40
primc.cpp:
Turbo Link  Version 7.1.30.1. Copyright (c) 1987, 1996 Borland International
```

Можем аналогичным образом попробовать поработать с последней (на момент написания данной книги) версией **BC++ 5.5** (входит в стандартную поставку **Borland C++ Builder 5**). Но это уже будет 32-разрядное приложение! Поэтому пока подождем.

6.1.2.6. Обеспечение корректности вычислений на C/C++ (Borland C++ 5.02) (вариант 1)

ПРИМЕР 6.2А.

Сделаем в нашем примере 6.2 изменения, направленные на то, чтобы корректно в C++ вводить числовые символы. Для этого введем новую функцию **int input (int& k);** В прилагаемом ниже файле изменения выделены жирным шрифтом.

Новый исходный текст программы PrimC.cpp

```
/* PrimC.cpp                Borland C++ 5.02
            Проект !!!!!! Application.exe === > DOS (standard) !!!!!!
                          asm+cpp файлы
            (c) Copyright 1993-1998-2000 by Голубь Н.Г.
                     Лекционный пример 6.2а.
                                              Авторское приложение
  к Методическим указаниям по курсам "Объектно-ориентированное программирование"
                        и "Организация и функционирование ЭВМ"
        Вычислить  x=(2*a+b*c)/(d-a);  int a,b,c,d,x;
            Корректный ввод числовых символов.
*/

#include <conio.h>
#include <fstream.h>
#include <limits.h>
```

```
const char* ErrorRange = "\n!!!!!!!!!!! Limits of int value !!!!!!!!!!!\n";
// Контроль диапазона для 16-разрядных знаковых чисел:
//              -32768<=a<=32767
//(c) Copyright 1998 by Артем Астафьев (629 группа ХАИ)
inline  int     test(long int a)
    {return ((a>>15)+1)&~1;}

int input (int& k) // КОРРЕКТНЫЙ ВВОД данных типа int
{ // Открытие своих вход-выходных потоков на консоль
    ifstream my_inp ("CON");
    ofstream my_out ("CON");
   long int temp;
   my_inp >> temp;
    switch (my_inp.rdstate())  // Анализ введенной информации
         { case ios::goodbit:
           case ios::eofbit :  // НЕТ ошибок ввода
                       if (test (temp))
                          { // Ошибка диапазона для переменной типа int
                            my_out << ErrorRange;
                            return 1;;
                            }
                       k = temp;
                       return 0;  // Все в порядке!
           case ios::failbit:            // Есть ошибки ввода
           case ios::badbit :
               my_out << "\n!!!!! Error input !!!!!\n";
               return 1;
         };
}

int     primC(int a, const int b, const int c, const int d)
{double z = (2.0*a+1.0*b*c)/(d-a);  //!!!!!!!!!! ВНИМАНИЕ !!!!!!!!!!!!!!!!
  if (z > SHRT_MIN && z < SHRT_MAX) return z; // test ЗДЕСЬ применять НЕЛЬЗЯ!!!
   else
   {cout << ErrorRange << "  Result x = " << z << endl;
     return SHRT_MIN; //-32768
    }
 }

extern "C"
{  void prim (void);}

int X,a;

void main(void)
{  char ch;
    const b=-333;
    const c= 1000;
    const d=-10;
   do {X = 0;  ·
       //clrscr();
       cout << "\n    x=(2*a+b*c)/(d-a);  int x,a,b=-333,c=1000,d=-10;" << endl;
       do{    cout<<"\n    Enter a [-32768..32767], a!= " << d << " ====> ";
              while (input(a));    //Вместо cin>> a1;  !!!!!! ВНИМАНИЕ !!!!!!!!!!!!!
            }while (d — a == 0 || test (long(d)-a));
      // Вы ввели следующее значение переменной
      cout << "\n You have entered the following value of a variable:  "
             << a << endl;
```

```
        X = primC(a, b, c, d);
        if (X!=SHRT_MIN)
         { cout << " Result (C++) x = " << X << endl;
           X = 0;
           prim();
           cout << " Result (ASM) x = " << X << endl;
         }
        cout << "\n\nExit? — (y/n)\n";
        ch = getch();
      } while (!(ch=='y' || ch =='Y'));
}
```

Результаты тестовой проверки:

```
============== Test # 1 =====================
        x=(2*a+b*c)/(d-a);  int x,a,b=-333,c=1000,d=-10;

        Enter a [-32768..32767], a!= -10 ====> -2

   You have entered the following value of a variable:  -2

!!!!!!!!!!! Limits of int value !!!!!!!!!!!
   Result x = 41625.5

Exit? — (y/n)

============== Test # 2 =====================
        x=(2*a+b*c)/(d-a);  int x,a,b=-333,c=1000,d=-10;

        Enter a [-32768..32767], a!= -10 ====> 333r

   You have entered the following value of a variable:  333
   Result (C++) x = 968
   Result (ASM) x = 968

Exit? — (y/n)
============== Test # 3 =====================

        x=(2*a+b*c)/(d-a);  int x,a,b=-333,c=1000,d=-10;
        Enter a [-32768..32767], a!= -10 ====> ti86
!!!!! Error input !!!!!
86ti
   You have entered the following value of a variable:  86
   Result (C++) x = 3466
   Result (ASM) x = 3466

Exit? — (y/n)
============== Test # 4 =====================
        x=(2*a+b*c)/(d-a);  int x,a,b=-333,c=1000,d=-10;
        Enter a [-32768..32767], a!= -10 ====> 32767
        Enter a [-32768..32767], a!= -10 ====> 32766
        Enter a [-32768..32767], a!= -10 ====> 32760
        Enter a [-32768..32767], a!= -10 ====> -32700
   You have entered the following value of a variable:  -32700
   Result (C++) x = -12
   Result (ASM) x = -12
Exit? — (y/n)
```

```
============== Test # 5 =====================
        x=(2*a+b*c)/(d-a);   int x,a,b=-333,c=1000,d=-10;
        Enter a [-32768..32767], a!= -10 ====> 32760
        Enter a [-32768..32767], a!= -10 ====> 32750
   You have entered the following value of a variable:  32750
   Result (C++) x = 8
   Result (ASM) x = 8
Exit? — (y/n)
```

 **ОБРАТИТЕ ВНИМАНИЕ**

на то, как по-разному обрабатываются буквы во введенном числе в тестах #2, #3. А в тестах #4, #5 показано, что ЗНАМЕНАТЕЛЬ **(d-a)** НЕ должен превышать диапазон [-32768..32767]. Почему это так важно? Что будет, если этого НЕ сделать? А будет вот что:

```
You have entered the following value of a variable:  32760
  Result (C++) x = -8
  Result (ASM) x = -8
```

Правильный ли мы получили результат?

Да, тяжелая это работа — из болота тащить бегемота.

6.1.2.7. Обеспечение корректности вычислений на C/C++ (Borland C++ 5.02) вариант 2

В предыдущем примере мы разобрались, как корректно вводить данные типа **int**. Сейчас мы обобщим этот опыт и на остальные ЦЕЛОЧИСЛЕННЫЕ переменные, то есть сделаем аналог примера 5.4b, но для ситуации **Cpp+ASM**. Это станет нашим шаблоном программы, который мы будем использовать в дальнейших примерах — тем самым мы можем забыть о проблемах с вводом ЦЕЛОЧИСЛЕННЫХ данных (**BC++ 5.x**).

 **ПРИМЕР 6.2B (АНАЛОГ ПРИМЕРА 5.4B).**

Написать программу сложения 8-, 16- и 32-разрядных ЦЕЛЫХ чисел. Сделаем программу на C++ более УНИВЕРСАЛЬНОЙ, используем такие *современные средства* программирования на C++, как *шаблоны* и *динамическое определение типа данных.*

Во-первых, создадим модуль **inputNum.h**, обеспечивающий корректность ввода ЛЮБЫХ ЦЕЛОЧИСЛЕННЫХ исходных данных (**unsigned char, unsigned int, char, int, long int**), используя параметризованную функцию:

```
template <class Type>
int input(const char* invite,Type& k);
```

Файл inputNum.h

```
#ifndef __INPUTNUM_H
#define __INPUTNUM_H
#include <fstream.h>

// (c) Copyright 1998-2001 by Голубь Н.Г.

typedef unsigned char byte;
typedef unsigned int word;
```

```
template <class Type>
int input(const char* invite,Type& k)
{
 //float x; // Для обеспечения проверки long int. Работает НЕ всегда !!!!!!!!! — см. тест #8
   long double x; // Работает НОРМАЛЬНО
   ifstream my_inp("CON");  // Создание СВОЕГО входного потока — ОБЯЗАТЕЛЬНО!!!
   ofstream my_out("CON");
   my_out << invite;
   my_out.flush();
   my_inp >> x;
   switch(my_inp.rdstate())
      {
           case ios::goodbit:
           case ios::eofbit:
             k = x;
             if (x!=k)     // ПРОВЕРКА допустимого диапазона
               { my_out << "\tError! Incorrect range for type " << typeid(k).name()
                        << endl;
                 return 1;
               }
             return 0; // ok!
           case ios::failbit:
           case ios::badbit:
             my_out<<"\tError! Incorrect format of number!!!\n";
             return 1;
      }
  return 1;
}
#endif
```

Во-вторых, создадим **asm**-файл, аналогичный приведенному в примере 5.4b, но с немного ИЗМЕНЕННЫМ для стыковки с C++ ИНТЕРФЕЙСОМ.

Исходный текст модуля Adda_asm.asm

```
          title Adda_asm.asm
;CopyRight by Голубь Н.Г., 1993-1997, 2001
          .MODEL Large, C
;            x=a+b    x,a,b:long int,
; Byte (unsigned char), char, int, Word (unsigned int)
          .data
          Extrn    C x:Dword,a:Dword,b:Dword   ; long int
; Byte (unsigned char)
          Extrn    C xS:Byte,aS:Byte,bS:Byte
          Extrn    C xB:Byte,aB:Byte,bB:Byte   ; char
          Extrn    C xI:Word,aI:Word,bI:Word   ; int
; Word (unsigned int)
          Extrn    C xW:Word,aW:Word,bW:Word
          .code
;============ long int
          Public  C addaL
addaL    proc     far
          mov   ax,WORD PTR a ; ax <=== мл. часть a
          mov   bx,WORD PTR a+2    ; bx <=== ст. часть a
          mov   cx,WORD PTR b       ; cx <=== мл. часть b
          mov   dx,WORD PTR b+2    ; dx <=== ст. часть b
```

```
        add   ax,cx ; <ax>:=<ax>+<cx>         мл.часть
        adc   bx,dx ; <bx>:=<bx>+<dx>+<CF>   ст.часть
        mov   WORD PTR x,ax    ; мл. часть x <=== <ax>
        mov   WORD PTR x+2,bx ; ст. часть x <=== <bx>
        ret
addaL    endp
;============ char
        Public  C addaS
addaS  proc far
        mov   aL, aS       ; aL <=== aS
        mov   cL, bS       ; cL <=== bS
        add   aL,cL        ; <aL>:=<aL>+<cL>
        mov   xS,aL        ; xS <=== <aL>
        ret
addaS    endp
;============ unsigned char (byte)
        Public  C addaB
addaB    proc      ar
        mov   aL, aB       ; aL <=== aB
        mov   cL, bB       ; cL <=== bB
        add   aL,cL        ; <aL>:=<aL>+<cL>
        mov   xB,aL        ; xB <=== <aL>
        ret
addaB    endp
;============ int
        Public  C addaI
addaI  proc far
        mov   ax, aI ; ax <=== aI
        mov   cx, bI ; cx <=== bI
        add   ax,cx  ; <ax>:=<ax>+<cx>
        mov   xI,ax  ; xI <=== <ax>
        ret
addaI  endp
;============ unsigned int (word)
        Public  C addaW
addaW  proc far
        mov   ax, aW ; ax <=== aW
        mov   cx, bW ; cx <=== bW
        add   ax,cx  ; <ax>:=<ax>+<cx>
        mov   xW,ax  ; xW <=== <ax>
        ret
addaW  endp
        end
```

В-третьих, сделаем главную программу на C++.

Исходный текст программы Pr6_2b.cpp

```
/* Pr6_2b.cpp              Borland C++ 5.02
      Проект !!!!!! Application.exe === > DOS (standard) !!!!!!
                    asm+cpp файлы
            (c) Copyright 1998-2001 by Голубь Н.Г.
                    Лекционный пример 6.2b.
                      x=a+b;
```

```
                a,b,x : int;
                a,b,x : char;
                a,b,x : long int;
                a,b,x : unsigned char (byte);
                a,b,x : unsigned int (word);
            Корректный ввод числовых символов.
*/

#include <iostream.h>
#include <conio.h>
#include <typeinfo.h>
#include "inputNum.h" // Ввод ЛЮБЫХ ЦЕЛЫХ чисел

const char* InputA = " Input a: ";
const char* InputB = " Input b: ";
const char* ResultC = "\n\n Result      in C++ = ";
const char* ResultASM = "\n\n Result      in ASM = ";
const char* CONTINUE = "\n\n Press any key to continue...";

// ВНЕШНИЕ ассемблерные функции
extern "C"
{
 void addaI (void);
 void addaS (void);
 void addaL (void);
 void addaB (void);
 void addaW (void);
}

void title(void)
{
 cout << "\n\tCalculate expression:   x = a + b\n";
}

void title(long int a,long int b)
{
 cout <<"\n\t  "<< a << " + " << b << " = ";
}

void PressAnyKey()
{cout <<  CONTINUE;
  getch();
}

char menu(void)
{
 cout<<"\n Select the type of arguments for calculation:\n\n"
    <<"\t 1 — Byte (1 byte without signum     — unsigned char)\n"
    <<"\t 2 — Char (1 byte with signum        — char )\n"
    <<"\t 3 — Word (2 bytes without signum   — unsigned int )\n"
    <<"\t 4 — Int  (2 bytes with signum       — int )\n"
    <<"\t 5 — long int  (4 bytes with signum — long int)\n"
    <<"\t 6 — Exit from this program...\n";
  return getch();
}
```

```
//————————— ГЛОБАЛЬНЫЕ переменные — передаются в Ассемблер
long int a,b,x;
int al,bl,xl;
byte aB,bB,xB;
char aS,bS,xS;
word aW,bW,xW;
//——————————————————————————————————

// ПАРАМЕТРИЗОВАННАЯ функция вычисления x=a+b
template <class Type>
int calculate(Type a,Type b, Type& Result)
{ //float x = (float)a+b;  — Вариант работает НЕ всегда!!!!!!!!!! — см. тест #9
  long double x = (long double)a+b;
  Result  = a+b;
  if (x != Result) // ПРОВЕРКА допустимого диапазона
   { cout << "\n!!!!!!! Result " << x << " is out range " << typeid(a).name();
     return 0;  // Error Range!
   }
  return 1;  // Ok!
}

void ProcByte()
{
 cout<<"\n Type of arguments BYTE (0..255)\n\n";
 while (input (InputA,aB));
 while (input (InputB,bB));
 title(aB,bB);
 if (calculate(aB,bB,xB))
  { cout << ResultC << (int)xB;
    xB=0; addaB();
    cout << ResultASM << (int)xB;
  }
}

void ProcChar()
{
 cout<<"\n Type of arguments CHAR (-128..127) \n\n";
 while (input (InputA,aS));
 while (input (InputB,bS));
 title(aS,bS);
 if (calculate(aS,bS,xS))
  {cout << ResultC << (int)xS;
    xS=0; addaS();
    cout << ResultASM << (int)xS;
  }
}

void ProcWord()
{
 cout<<"\n Type of arguments WORD (0..65535)\n\n";
 while (input (InputA,aW));
 while (input (InputB,bW));
 title(aW,bW);
 if (calculate(aW,bW,xW))
  {cout <<  ResultC << xW;
    xW=0; addaW();
    cout << ResultASM << xW;
  }
}
```

```
void ProcInt()
{
 cout<<"\n Type of arguments INT (-32768..32767)\n\n";
 while (input (InputA,al));
 while (input (InputB,bl));
 title(al,bl);
 if (calculate(al,bl,xl))
  {cout << ResultC << xl;
   xl=0; addal();
   cout << ResultASM << xl;
  }
}

void ProcLongInt()
{
 cout<<"\n Type of arguments  LONG INT (-2147483648..2147483647)\n\n";
 while (input (InputA,a));
 while (input (InputB,b));
 title(a,b);
 if (calculate(a,b,x))
  {cout << ResultC << x;
   x=0; addaL();
   cout << ResultASM << x;
  }
}

int main(void)
{
 int t=0;
 char point;
 do{
 clrscr();
 title();
 point = menu();
 switch(point)
 {
  case "1":ProcByte();break;
  case "2":ProcChar();break;
  case "3":ProcWord();break;
  case "4":ProcInt(); break;
  case "5":ProcLongInt(); break;
  case "6": return 0;
 }
 cout << "\n============= test #" << ++t << " =============\n";
 PressAnyKey();
 }while (point != "6");
}
```

Проверим нашу программу.

```
          Calculate expression:   x = a + b

 Select the type of arguments for calculation:

          1 — Byte (1 byte without signum    — unsigned char)
          2 — Char (1 byte with signum        — char )
          3 — Word (2 bytes without signum   — unsigned int )
          4 — Int  (2 bytes with signum       — int )
          5 — long int  (4 bytes with signum — long int)
          6 — Exit from this program...

 Type of arguments  BYTE (0..255)

 Input a: 222
 Input b: 111
          222 + 111 =

!!!!!!! Result 333 is out range unsigned char

==================== test #1 =====================

 Press any key to continue...
          Calculate expression:   x = a + b

 Select the type of arguments for calculation:

          1 — Byte (1 byte without signum    — unsigned char)
          2 — Char (1 byte with signum        — char )
          3 — Word (2 bytes without signum   — unsigned int )
          4 — Int  (2 bytes with signum       — int )
          5 — long int  (4 bytes with signum — long int)
          6 — Exit from this program...

 Type of arguments  BYTE (0..255)

 Input a: -11
          Error! Incorrect range for type unsigned char
 Input a: 22
 Input b: 33de

          22 + 33 =

 Result      in C++ = 55

 Result      in ASM = 55

==================== test #2 =====================

 Press any key to continue...

          Calculate expression:   x = a + b

 Select the type of arguments for calculation:

          1 — Byte (1 byte without signum    — unsigned char)
          2 — Char (1 byte with signum        — char )
          3 — Word (2 bytes without signum   — unsigned int )
          4 — Int  (2 bytes with signum       — int )
          5 — long int  (4 bytes with signum — long int)
          6 — Exit from this program...
```

```
 Type of arguments CHAR (-128..127)

 Input a: -22
 Input b: -222
           Error! Incorrect range for type char
 Input b: -120

              -22 + -120 =

!!!!!!! Result -142 is out range char
====================== test #3 ========================

 Press any key to continue...
           Calculate expression:    x = a + b

 Select the type of arguments for calculation:

             1 - Byte (1 byte without signum    - unsigned char)
             2 - Char (1 byte with signum        - char )
             3 - Word (2 bytes without signum   - unsigned int )
             4 - Int  (2 bytes with signum       - int )
             5 - long int  (4 bytes with signum - long int)
             6 - Exit from this program...

 Type of arguments CHAR (-128..127)

 Input a: -128
 Input b: 127

              -128 + 127 =

 Result        in C++ = -1

 Result        in ASM = -1
===================== test #4 ========================

 Press any key to continue...
           Calculate expression:    x = a + b

 Select the type of arguments for calculation:

             1 - Byte (1 byte without signum    - unsigned char)
             2 - Char (1 byte with signum        - char )
             3 - Word (2 bytes without signum   - unsigned int )
             4 - Int  (2 bytes with signum       - int )
             5 - long int  (4 bytes with signum - long int)
             6 - Exit from this program...

 Type of arguments WORD (0..65535)

 Input a: 44444
 Input b: -4
           Error! Incorrect range for type unsigned int
 Input b: -11111
           Error! Incorrect range for type unsigned int
 Input b: 11111

              44444 + 11111 =

 Result        in C++ = 55555

 Result        in ASM = 55555
```

```
====================== test #5 ========================

 Press any key to continue...
         Calculate expression:   x = a + b

 Select the type of arguments for calculation:

          1 - Byte (1 byte without signum    - unsigned char)
          2 - Char (1 byte with signum        - char )
          3 - Word (2 bytes without signum   - unsigned int )
          4 - Int  (2 bytes with signum       - int )
          5 - long int  (4 bytes with signum - long int)
          6 - Exit from this program...

 Type of arguments INT (-32768..32767)
 Input a: -423
 Input b: 4355

           -423 + 4355 =

 Result       in C++ = 3932

 Result       in ASM = 3932
====================== test #6 ========================

 Press any key to continue...
         Calculate expression:   x = a + b

 Select the type of arguments for calculation:

          1 - Byte (1 byte without signum    - unsigned char)
          2 - Char (1 byte with signum        - char )
          3 - Word (2 bytes without signum   - unsigned int )
          4 - Int  (2 bytes with signum       - int )
          5 - long int  (4 bytes with signum - long int)
          6 - Exit from this program...

 Type of arguments  LONG INT (-2147483648..2147483647)

 Input a: 2222222
 Input b: 111111

           2222222 + 111111 =

 Result       in C++ = 2333333
 Result       in ASM = 2333333

====================== test #7 ========================

 Press any key to continue...
         Calculate expression:   x = a + b

 Select the type of arguments for calculation:

          1 - Byte (1 byte without signum    - unsigned char)
          2 - Char (1 byte with signum        - char )
          3 - Word (2 bytes without signum   - unsigned int )
          4 - Int  (2 bytes with signum       - int )
          5 - long int  (4 bytes with signum - long int)
          6 - Exit from this program...
```

```
 Type of arguments  LONG INT (-2147483648..2147483647)
 Input a: 1111111111
 Input b: 1111111111
            1111111168 + 1111111168 = ?????????????
!!!!!!! Result 2.22222e+09 is out range long
===================== test #8 =======================
 Press any key to continue...
        Calculate expression:   x = a + b
 Select the type of arguments for calculation:
          1 - Byte (1 byte without signum    - unsigned char)
          2 - Char (1 byte with signum        - char )
          3 - Word (2 bytes without signum   - unsigned int )
          4 - Int  (2 bytes with signum       - int )
          5 - long int  (4 bytes with signum - long int)
          6 - Exit from this program...
 Type of arguments  LONG INT (-2147483648..2147483647)
 Input a: 1111111111111
        Error! Incorrect range for type long
 Input a: 11111111
 Input b: 22222222
          11111111 + 22222222 =
!!!!!!! Result 3.33333e+07 is out range long  ?????????????
===================== test #9 =======================
 Press any key to continue...
```

— После изменений float x; на long double x; —————

```
        Calculate expression:   x = a + b
 Select the type of arguments for calculation:
          1 - Byte (1 byte without signum    - unsigned char)
          2 - Char (1 byte with signum        - char )
          3 - Word (2 bytes without signum   - unsigned int )
          4 - Int  (2 bytes with signum       - int )
          5 - long int  (4 bytes with signum - long int)
          6 - Exit from this program...
 Type of arguments  LONG INT (-2147483648..2147483647)
 Input a: 1111111111
 Input b: 1111111111

          1111111111 + 1111111111 =
!!!!!!! Result 2.22222e+09 is out range long
===================== test #1 =======================
 Press any key to continue...
        Calculate expression:   x = a + b
 Select the type of arguments for calculation:
```

```
            1 — Byte (1 byte without signum      — unsigned char)
            2 — Char (1 byte with signum         — char )
            3 — Word (2 bytes without signum     — unsigned int )
            4 — Int  (2 bytes with signum        — int )
            5 — long int  (4 bytes with signum   — long int)
            6 — Exit from this program...

  Type of arguments  LONG INT (-2147483648..2147483647)

  Input a: 11111111
  Input b: 22222222

             11111111 + 22222222 =

  Result       in C++ = 33333333
  Result       in ASM = 33333333
==================== test #2 =====================

  Press any key to continue...
```

Как видите, сейчас все нормально.

Вопросы.

1. Как вы думаете, почему компьютер именно так воспринял наши данные в тестах #8 и #9? **Ключ** к ответу на этот вопрос — опять МАШИННОЕ представление, на этот раз, вещественных данных — см. п.2.3.2.
2. Можно ли было в данной ситуации использовать описание **double** x?

6.1.2.8. Поддержка кириллицы в консольных приложениях на C/C++

Приведенный выше шаблон хорош всем, за исключением того, что писать программные пояснения и приглашения надо с использованием латиницы (в данном случае английского языка), что не для всех, к сожалению, подходит. Все-таки русский (украинский, белорусский, болгарский) интерфейс как-то роднее. Для облегчения этой ситуации мой ученик, а ныне коллега Лучшев П.А. предложил маленький файл ПЕРЕКОДИРОВКИ кириллицы из **Win** в **DOS**.

Исходный текст модуля перекодировки кириллицы convert.h

```
///////////////////////////////////////////////////////////////////
//
// Version 1.0b — Copyright by Лучшев П.А. 10.09.1999
// convert.h : Конвертирование кодировки Win (System) в DOS (Terminal)
//  для использования кириллицы в консольных приложениях
//          и    Application.exe === > DOS (standard)
//            работает в Borland C++ 4.x, 5.x
//
//       Использование:
// cout << TXT("АБВГД"); <==== в этом случае НЕ всегда
//                               преобразования КОРРЕКТНЫ
// char* str = TXT("АБВГД"); <==== так получается лучше с
//                               преобразованиями символов
//
// Примечание:  строка должна быть стандартной и заканчиваться символом "\0"
//
///////////////////////////////////////////////////////////////////
```

```
#ifndef _CONVERT_H_
#define _CONVERT_H_

char* WtoD(char* str)
{ for (char* tmp = str; *tmp != 0; tmp++)
   {      if ((*tmp >= "A") && (*tmp <= "п")) (*tmp) -= 64;
           else if ((*tmp >= "р") && (*tmp <= "я")) (*tmp) -= 16;
   }
  return str;
}

#define TXT(a) WtoD(a)

#endif // _CONVERT_H_
```

Покажем, как этот модуль перекодировки можно применять, на примере нашего всепогодного модуля ввода.

Исходный текст модуля inputNum.h (с поддержкой кириллицы)

```
#ifndef __INPUTNUM_H
#define __INPUTNUM_H
#include <fstream.h>
#include "convert.h"  // Перекодировка КИРИЛЛИЦЫ Win—>DOS
// (c) Copyright 1998-2001 by Голубь Н.Г.
typedef unsigned char byte;
typedef unsigned int word;
const char* ERROR = TXT("\tОшибка! НЕ корректен ");
const char* RANGE_TYPE = TXT("диапазон для типа ");
const char* FORMAT = TXT("формат числа!!!\n");
template <class Type>
int input(const char* invite,Type& k)
{
  //float x; // Для обеспечения проверки long int. Работает НЕ всегда!!!!!!!!!!
  long double x;          // Работает НОРМАЛЬНО
  ifstream my_inp("CON");
  ofstream my_out("CON");
  my_out << invite;
  my_out.flush();
  my_inp >> x;
  switch(my_inp.rdstate())
   {
     case ios::goodbit:
     case ios::eofbit:
        k = x;
     if (x!=k)     // ПРОВЕРКА допустимого диапазона
         { my_out << ERROR << RANGE_TYPE << typeid(k).name()
                         << endl;
           return 1;
         }
     return 0; // ok!
     case ios::failbit:
     case ios::badbit:
        my_out << ERROR << FORMAT;
        return 1;
   }
  return 1;
}
#endif
```

Полностью русскоязычный вариант примера 6.2b находится в поддиректории **\Prб_2rus**.

Модуль перекодировки кириллицы прекрасно работает и в среде **Visual C++ 5.0** и **ниже** (см. поддиректорию **\VCpp**), а вот в **Visual C++ 6.0** работать уже НЕ хочет!!! Бывает и так...

6.2. Встроенный Ассемблер

Для начинающих осваивать язык Ассемблера встроенный Ассемблер — это САМЫЙ ЛЕГКИЙ путь. Но в то же время встроенный Ассемблер — это НЕ полноценный Ассемблер, а суррогат. Например, встроенный Ассемблер можно сравнить с порошковым молоком (бывает ОЧЕНЬ качественное!), но все же НЕ натуральное молоко...

Хотя, с другой стороны, **профессионалы-программисты** для своих нужд очень часто пользуются **ассемблерными вставками**. Они ОЧЕНЬ хорошо знают, что им нужно, и как повысить качество своего продукта. Мы тоже, со временем, будем это делать.

Так что НЕ будем НИЧЕГО и НИКОГО осуждать, а попробуем разобраться сами...

6.2.1. Ассемблер, встроенный в программу на языке Pascal

Поскольку **Borland/Turbo Pascal—7.0x** разрабатывался в период с 1983 по 1992 годы, понятно, что он в принципе НЕ может поддерживать машинные инструкции семейства **Pentium** и выше — см. табл. 3.1. Более того, поскольку эта среда разработки является 16-разрядной, НЕ поддерживаются инструкции, связанные с 32-разрядным программированием (**зато это поддерживает Delphi!**).

Мы пока остановимся на встроенном Ассемблере **Borland/Turbo Pascal—7.0x**. Паскаль позволяет *комбинировать* код, используя встроенный Ассемблер, что является ОЧЕНЬ привлекательным для новичков, которые еще НЕ совсем разобрались в тонкостях программирования, например, в передаче параметров — см. пример 6.3. Кроме того, можно получать отладочную информацию в нормальном человеческом виде, а НЕ во внутреннем (машинном) представлении.

Рассмотрим основные особенности встроенного Ассемблера.

1. Встроенный Ассемблер реализует команды процессоров i8086 (по умолчанию) и i80286 (ТОЛЬКО при наличии директивы **{$G+}**), а также команды сопроцессоров i8087 (при наличии директив **{$N+} {$E+}**) и i80287 (при наличии директив **{$G+} {$N+} {$E+}**). Это как раз те базовые команды, которые мы сейчас изучаем.
2. Для всех объявлений переменных и меток, а также для комментариев используется синтаксис языка Паскаль.
3. К встроенному Ассемблеру обращаются через оператор ASM:

   ASM
   Команды_Ассемблера
   END;
4. Встроенный Ассемблер НЕ может ИЗМЕНЯТЬ регистры **BP, SP, SS, DS**.
5. Встроенный Ассемблер поддерживает только три директивы Ассемблера: **DB, DW, DD**.
6. Встроенный Ассемблер вычисляет все выражения как 32-битные целые выражения:

   [-2147483648 ..4294967295]

7. Значения выражений с плавающей точкой НЕ поддерживаются.
8. Выражения Паскаля допускают только десятичную или шестнадцатеричную (префикс **$**) нотацию целочисленных констант.
9. Строковые константы поддерживаются в стиле Паскаль.
10. Встроенный Ассемблер всегда интерпретирует параметр **VAR** как 32-битный указатель: **ES:[*регистр*]**. Поэтому надо сначала параметр **VAR** через команду **LES** загрузить в регистр:

    LES *регистр, VAR-параметр,*

    а потом уже использовать его как **ES:[*регистр*]**.
11. Паскаль имеет директиву **ASSEMBLER**, которая во многом сравнима с директивой EXTERNAL. Эта директива позволяет строить на Паскале ассемблерные процедуры и функции по тем же правилам, что и внешние процедуры и функции.

ПРИМЕР 6.3.

Реализовать функцию **Sum** = x + y для данных типа **integer**.

Это можно сделать на чистом Паскале — см. вариант 1, с применением встроенного Ассемблера (см. варианты 2 и 3), а также с помощью ассемблерной функции — вариант 4. Варианты 3 и 4 используют специальную переменную Паскаля **@Result** для возврата из функции значения. **Вариант 4 — самый близкий вариант к чистому Ассемблеру**. Именно его рекомендуется использовать при отладке тем, кто *изучает* Ассемблер. В этом примере есть некоторые тонкости, связанные с возвратом параметров. Их мы рассмотрим позднее — см. п. 9.1.2.3.

Исходный текст программы AsmInPas.pas

```
Program AsmInPas;
{------------------------------------------------------------
          Sum = x + y для данных типа integer.
     Входные данные и результат на ДОПУСТИМЫЙ диапазон
          !!!!!!!!!!! НЕ ПРОВЕРЯЮТСЯ !!!!!!!!!!!
       Лекционный пример 6.3. Встроенный Ассемблер.
          CopyRight by Голубь Н.Г., 1998-2001
------------------------------------------------------------}
Var
  x,y,s   : Integer;

{Вариант 1 — ПАСКАЛЬ}
Function Sum1 (x,y:Integer):Integer;
Begin
  Sum1 := x+y
End;
{Вариант 2 — Встроенный Ассемблер — передача параметров по ЗНАЧЕНИЮ}
Function Sum2 (x,y:Integer):Integer;
Begin
 ASM
    mov ax,x
    add  ax,y
    mov @Result,ax
 END
```

```
End;
{Вариант 3 — Встроенный Ассемблер — передача параметров по ССЫЛКЕ}
Function Sum3 (Var x,y:Integer):Integer;
Begin
  ASM
     les   bx,x
{КОСВЕННАЯ адресация — берем значение по адресу}
     mov ax,es:[bx]
     les   bx,y
     add   ax,es:[bx]
     mov  @Result,ax
  END
End;
{Вариант 4 — Ассемблерная функция}
Function Sum4 (x,y:Integer):Integer; ASSEMBLER;
ASM
     mov ax,x
     add  ax,y
End;
begin
   Writeln(' Вычислить:       Sum = x + y для данных типа Integer;'');
   Write('Введите значение x  ');
   Readln(x);
   Write('Введите значение y  ');
   Readln(y);
   s := Sum1(x,y);
   writeln ('ПАСКАЛЬ: Sum = ', x, ' + ', y, ' = ',s);
   s := 0; {для ЧИСТОТЫ эксперимента}
   s := Sum2(x,y);
   writeln (' Встроенный Ассемблер');
   writeln ( — передача параметров по ЗНАЧЕНИЮ: Sum = , x, + , y, = ,s);
   s := 0;
   s := Sum3(x,y);
   writeln (' — передача параметров по ССЫЛКЕ: Sum = , x, + , y,=,s);
   s := 0;
   s := Sum4(x,y);
   writeln ('Ассемблерная функция: Sum = ', x, ' + ', y, ' = ',s);
   writeln ('    Нажмите любую клавишу.........');
   Readln
end.
```

6.2.2. Ассемблер, встроенный в программу на языке C/C++

Поскольку язык C++ и компиляторы, его поддерживающие, продолжают развиваться, команды, которые поддерживает встроенный Ассемблер, существенно зависят от версии компилятора и от его разработчиков.

6.2.2.1. Интегрированная среда программирования Borland C++

Встроенный Ассемблер компилятора **Borland/Turbo C++ 3.1** практически НЕ отличается от встроенного Ассемблера **Borland/Turbo Pascal—7.0x**, поскольку время окончания разработки одинаковое. А вот старшие версии, конечно, поддерживают все большее число команд и архитектур процессоров. Синтаксис функций, использующих встроенный Ассемблер, тоже напоминает стиль языка Паскаль, но, как принято в C/C++, он более лаконичный. Например, наш пример 6.1 с ассемблерными вставками будет иметь следующий вид:

```
int funcAsmI (int x)
{ asm
    { mov ax,x
      cwd
      mov cx,3
      idiv cx
    }
 }
unsigned int funcAsmU (unsigned int x)
{ asm
    { mov ax,x
      xor dx,dx
      mov cx,3
      div cx
    }
 }
```

Полностью программу можно найти в прилагаемых к данной книге электронных материалах — файл **...\TASM\Part1\6\Pr6_1\ BCpp\cpp_asm_i.cpp**.

6.2.2.2. Интегрированная среда программирования Microsoft Visual C++ 6.0

Здесь необходимо учитывать, что **компилятор VC++ является 32-разрядным**. Поэтому, чтобы СОХРАНИТЬ исходную постановку задачи, наш пример 6.1 немного видоизменится:

1) описания данных типа **int** нужно ВСЮДУ заменить на тип **short int**;

2) оператор **asm** должен быть с ДВУМЯ символами подчерка **__asm**.

Исходный текст программы 1.cpp

```
// 1.cpp : Defines the entry point for the console application.
//
/*          (c) Copyright 2001 by Голубь Н.Г.
                  Встроенный Ассемблер в VC++ 6.0
                Лекционный пример 6.1.
      ВНИМАНИЕ!!! Диапазон вводимых данных !!!!!!!!!!!!!!!!!!!
*/
#include "stdafx.h"
#include "iostream.h"
#include "typeinfo.h"

short int funcCi(short int x)
{ return x/3;}

short int funcCu(unsigned short int x)
{ return x/3;}

short int funcAsmI (short int x)
{ __asm
    { mov ax,x
      cwd
      mov cx,3
      idiv cx
    }
 }
```

```
unsigned short int funcAsmU (unsigned short int x)
{ __asm
    { mov ax,x
      xor dx,dx
      mov cx,3
      div cx
    }
 }

int main(int argc, char* argv[])
{short int a;
 unsigned short b;
 for (;;)
  { cout << "???? " << typeid(a).name() << " a =======> "; cin >> a;
    cout << "VC++ 6.0: " << a << "/3 = " << funcCi(a) << endl;
    cout << "     Assembler: " << funcAsmI(a) << endl << endl;
    cout << "???? " << typeid(b).name() << " b =======> "; cin >> b;
    cout << "VC++ 6.0: " << b << "/3 = " << funcCu(b) << endl;
    cout << "     Assembler: " << funcAsmU(b) << endl;
     cout << "Exit     Ctrl-C\n\n";
  }
   return 0;
}
```

Результат тестирования:

```
???? short a =======> 11111
VC++ 6.0: 11111/3 = 3703
      Assembler: 3703

???? unsigned short b =======> 55555
VC++ 6.0: 55555/3 = 18518
      Assembler: 18518

Exit      Ctrl-C
???? short a =======> -11111
VC++ 6.0: -11111/3 = -3703
      Assembler: -3703

???? unsigned short b =======> -3333
VC++ 6.0: 62203/3 = 20734
      Assembler: 20734
Exit      Ctrl-C
```

И опять мы спровоцировали НАРУШЕНИЕ ДИАПАЗОНА, но в данном случае число помещается в 16 бит. Попробуйте ввести, например, число 111111. Что будет?

6.3. Введение в отладку программ

Если бы строители строили здания так же, как программисты пишут программы, первый залетевший дятел разрушил бы цивилизацию.

Второй закон Вейнберга (Артур Блох "Закон Мерфи")

Пустяковые вопросы решаются быстро; важные — никогда не решаются.

Закон Грехэма (Артур Блох "Закон Мерфи")

Мы все из своего жизненного опыта знаем, что ошибки встречаются на каждом шагу. Они бывают разные, бывают **критические**, когда надо что-то срочно делать, чтобы исправить ситуацию, а бывают вроде **предупреждений**: реагируешь, молодец, не реагируешь — тоже до поры до времени можно ничего не предпринимать, авось, пронесет...

То же самое происходит и при разработке программ — ведь это результат человеческой деятельности! В жизни люди обычно НЕ любят, когда их, словно котят, тыкают носом в ошибки (некоторые даже считают, что НИКОГДА их НЕ совершали, — святая простота!..). В программировании ошибки видны достаточно хорошо, хотя, может быть, и НЕ сразу, и НЕ всем (как и в жизни!). Пользователи начинают либо теребить разработчика программного продукта, либо искать замену неудачной программе, — работать ведь надо! Либо, если есть время и желание, пытаются все же обойти опасные места.

Поэтому принципиально важно по возможности еще ДО и в ПРОЦЕССЕ написания программы ОЧЕНЬ тщательно продумать, что же должна делать ваша программа и какие возможны ошибочные ситуации (но, как и во всяком деле, надо соблюдать, конечно, меру!). Обычно же происходит наоборот — вначале программировать, потом думать. Автор этих строк очень хорошо в свое время это прочувствовала при реальной разработке компилятора, поэтому прямо с первых шагов обучения начала вас тыкать носом (см. пример 5.4а), прекрасно понимая, что кому-то это может ОЧЕНЬ не понравиться. Ведь хочется поскорее получить результат (*а после нас хоть потоп*!), а тут надо еще думать о каком-то пользователе (ставить защиту от дурака), код растет, время поджимает... И вообще, *нельзя объять необъятное...*

Почитаем, что думают профессионалы. ***Большая программная система никогда не может быть отлажена до конца, даже после многих лет тестирования и использования.*** *До конца могут быть отлажены только маленькие программы, большие же и сложные не могут. Слишком много путей предусматривает общий алгоритм программы, слишком много возможных вариантов вводимых данных или действий пользователей. Даже сотни лет продолжающегося тестирования — если бы, конечно, это стало возможным, — не хватило бы для проверки всех возможных ветвей большой, сложной программы. И несмотря на это, находятся люди, всерьез говорящие о "программном обеспечении, свободном от ошибок!"*, — Джозеф Фокс [Л1-10]. Отсюда понятно, почему такие большие программные системы, как Microsoft Windows, Microsoft Visual Studio все время выходят с обновлениями (сервис-паками). Но прогресс и тут не стоит на месте: в помощь программистам ими же самими разрабатываются шаблоны (**Templates**), визуальные средства проектирования и разработки программ, новые технологии программирования.

Но все же **основные ошибки** можно предвидеть, а для этого надо знать, какие они бывают — см. п. 6.3.1, и **НЕ допустить их появления еще на стадии написания программы**. Конечно, здесь ОЧЕНЬ большую роль играют опыт и культура программирования. На протяжении всей книги на этом тоже будет *постоянно* акцентироваться ваше внимание. Лучше сразу привыкать писать грамотные программы — это для всех, в конечном счете, обойдется дешевле!

Хороших книг по современным методам отладки, к сожалению, совсем немного. Тем более приятно, что в 2001 году вышли сразу две замечательные книги [Л7-7, Л7-9], которые прекрасно дополняют друг друга. Если будет возможность, загляните в них — ОЧЕНЬ советую.

6.3.1. Категории ошибок в программах

Воспользуемся классификацией ошибок, предложенных Крисом Х. Паппасом и Уильямом Х. Мюррей III [Л7-7].

- **Синтаксические ошибки**. Они возникают на этапе компиляция из-за НЕЗНАНИЯ программистом каких-либо конструкций языка программирования или просто по невнимательности. С этими ошибками успешно справляются сами компиляторы. Поэтому такие ошибки хоть и раздражают, но считаются вполне безобидными. Еще компиляторы могут выдавать *предупреждения*, которые многие опытные программисты (и автор данной книги) склонны рассматривать как потенциальную угрозу логических ошибок. Бывают очень продвинутые компиляторы, очень редко ошибающиеся в своих предупреждениях (например, **Microsoft Visual C++ 6.0, Borland C++ 5.02**).
- **Ошибки компоновщика**. Это НАШ класс ошибок! Возникают они чаще всего от неправильно оформленного интерфейса стыковки разных модулей. А если система работает с модулями на каком-то одном языке, наиболее вероятной причиной ошибок является то, что НЕПРАВИЛЬНО указаны ДИРЕКТОРИИ библиотек и прочих подключаемых файлов.
- **Ошибки времени выполнения**. Эти ошибки ведут к аварийному преждевременному завершению программы или зацикливанию:
  - *аппаратно определяемые ошибки*: "Деление на НОЛЬ", "Попытка вычислить логарифм НУЛЯ или ОТРИЦАТЕЛЬНОГО числа", нарушение защиты памяти, ошибки устройства и проч.;
  - *системные ошибки*: неудачная файловая операция, переполнение очереди сообщений;
  - *чисто программные ошибки*: выход индекса за границы массива, удаление элемента из пустой очереди, выход на ветвь решения, которая, по идее, НИКОГДА НЕ должна была появиться и т.п.;
  - *ошибки, специфичные для приложений*: например, неверный формат ввода (в C++ стандартный поток **cin** в этом случае ...срывается..., именно поэтому мы и сделали СВОЙ входной поток **my_inp** — см. пример 6.2b, файл **inputNum.h**).
- **Логические ошибки**. Это *самый сложный вид ошибок*, поскольку они проявляются НЕ сразу, их предвидение напрямую зависит от квалификации и культуры программиста, а также от правильности постановки задачи заказчиком. *Именно для поиска и локализации таких ошибок существует процесс отладки.*

6.3.2. Процесс отладки

Отладка (**Debug, Debugging**) по терминологии фирмы IBM — обнаружение, локализация и устранение ошибочных операций в программах или отказов в компьютере.

Процесс отладки — очень интересный, *творческий* процесс как для начинающих, так и для опытных программистов. Вначале вы, как первооткрыватель, лезете в нехоженые джунгли своей программы и учитесь смотреть по сторонам (на содержимое разных регистров, на поведение своих, казалось, ручных переменных и проч.). Потом вы учитесь укрощать переменные, менять содержимое регистров и чувствовать, что компьютер, в общем-то, както к вам привыкает, да и вы к нему тоже (например, "сбоить" начинает все меньше и меньше). Набравшись опыта и знаний, вы начинаете смело провоцировать ошибки в своей программе, как опытный дрессировщик провоцирует, скажем, тигра — все вокруг ужасаются, но вы-то знаете, что делать с тигром...

Воспользуемся советами **Джона Роббинса** [Л7-9] по процессу отладки. Он выделяет **девять шагов**, которые *"интуитивно используют все великие разработчики"*. Попытаемся и мы пройтись вместе с Д. Роббинсом по ним и проиллюстрируем их на примере. Нам уже будет легче, потому что кое-что мы с вами уже делали. Значит, у нас уже есть кое-какой опыт. Итак, в добрый путь!

ПРИМЕР 6.4.

Вычислить арифметическое выражение **x = (2*c-d)/(a+b)**, где **x,a,b,c,d: integer**.

Реализуем вариант стыковки **Pascal+Assembler**, воспользовавшись рассуждениями и заготовками (шаблонами) примера 5.4b, а также постараемся сразу же при написании программы предусмотреть, что ЗНАМЕНАТЕЛЬ **a+b** НЕ ДОЛЖЕН БЫТЬ РАВЕН НУЛЮ. Получим, например, следующую программу на Паскале и **asm**-модуль.

Исходный текст программы Pr6_4.pas

```
Program Pr6_4;
{————————————————————————————————————————————
          x = (2*c-d)/(a+b)
        a,b,c,d,x : Integer;
   Лекционный пример 6.4. — вариант 1
      Вариант с КОНТРОЛЕМ данных
   CopyRight by Голубь Н.Г., 2001
—————————————————————————————————————————————}
{$I Pr6_4asm}   {вызов ASM-модуля Pr6_4asm.obj}
{$f+}
Uses CRT, InputNum;
Const
  {Символьные константы}
  inv2='Результат ';
  invAi='A (Integer)';
  invBi='B (Integer)';
  invCi='C (Integer)';
  invDi='D (Integer)';
  {Допустимый диапазон для вводимых данных и результата}
  IntMin=-32768;
  IntMax= 32767;
```

```
Label L1;
var  {Описание ГЛОБАЛЬНЫХ переменных}
  aL,bL,cL,dL,xL   : LongInt;
  x,a,b,c,d: Integer;
  ch       : Char;

{Внешняя процедура на Ассемблере}
Procedure Fasm(var x:Integer; a,b,c,d:Integer); external;

{Функция вычисления x = (2*c-d)/(a+b) с проверкой РЕЗУЛЬТАТА на допустимый диапазон}
function F (a,b,c,d:Integer; Var x:Integer; Min,Max:Longint) : Boolean;
Var x1 : Real;
    temp : LongInt;
    s1 : String;
Begin
  F:=True;  {Ошибок НЕТ}
  temp := a+b;
  if temp = 0 then
    begin
      Writeln ('!!!!! Знаменатель равен НУЛЮ — вычисления невозможны !!!!!!');
      F:=False; {Признак ошибки}
      Exit;
    end;
  x1 := (2.*c-d)/temp;
  if (x1>=Min)and(x1<=Max) then x:=trunc(x1)
   else
     Begin
       s1:=inv2+inv3;  {Формирование сообщения об ошибке}
       Writeln(s1,Min,'..',Max,']!!!!');
       {Вывод полученного значения, выходящего ЗА ДОПУСТИМЫЙ диапазон}
       Writeln(x1);
       Writeln(inv1);
       F:=False; {Признак ошибки}
       Exit;
     End
End;

Procedure Expl;  {Integer}
Label L1;
Begin
  Writeln(' Вычислить:     x = (2*c-d)/(a+b); a,b,c,d,x : Integer;');
 L1:
   InputNumber(aL,invAi,IntMin,IntMax);
   InputNumber(bL,invBi,IntMin,IntMax);
   InputNumber(cL,invCi,IntMin,IntMax);
   InputNumber(dL,invDi,IntMin,IntMax);
   a:=aL; b:=bL;c:=cL;d:=dL;
   if F(a,b,c,d,x,IntMin,IntMax) then Writeln ('ПАСКАЛЬ:   x=',x)
{Результат выходит за диапазон или деление на 0, повторяем ввод данных}
     else goto L1;
   x:=0;
   Fasm(x,a,b,c,d);
   writeln ('АССЕМБЛЕР: x=',x);
End;
```

```
begin
  Repeat
    ClrScr;
    Expl;
    Writeln('Повторим? (y/n)');
    ch:=ReadKey;
  Until (ch='n') or (ch='N');
end.
```

Исходный текст модуля Pr6_4asm.asm

```
        title Pr6_4asm.asm
; CopyRight by Голубь Н.Г., 2001
        .MODEL Large, Pascal
        ;x = (2*c-d)/(a+b) — вариант 1
        .data
        ;    x,a,b,c,d:Integer
        Extrn    x:word,a:word,b:word,c:word,d:word
        .code
        Public     Fasm
Fasm    proc       far
;——— знаменатель
        mov        bx, b      ; bx <=== b
        add        bx, a      ; <bx>=<bx>+a
; ——— числитель
        mov        ax, 2      ; ax <=== 2
        Imul       c          ; <dx:ax>=2*c
        sub        ax,d       ; <ax>=<ax>-d
        CWD                   ; <dx:ax> <=== <ax>
; ——— Результат
        Idiv       bx         ; <ax>=<dx:ax>/<bx>
        mov        x,ax       ; x <=== <ax>
        ret
Fasm    endp
   end
```

1. **Воспроизведение ошибки**. Это *наиболее критический шаг*. Иногда сделать его достаточно сложно или даже невозможно (недостаточно опытные программисты могут, как уже указывалось, ошибок вообще НЕ видеть!). А раз НЕТ ошибки, то и устранять НЕЧЕГО. Итак, запустим нашу программу на счет и получим следующий протокол:

```
——————— Test #1 ——————————————
Вычислить:       x = (2*c-d)/(a+b); a,b,c,d,x : Integer;
Введите значение A (Integer)===>100
Введите значение B (Integer)===>-99
Введите значение C (Integer)===>10000
Введите значение D (Integer)===>1
ПАСКАЛЬ:    x=19999
АССЕМБЛЕР: x=19999
Повторим? (y/n)
 Вычислить:       x = (2*c-d)/(a+b); a,b,c,d,x : Integer;
Введите значение A (Integer)===>1000
Введите значение B (Integer)===>999
Введите значение C (Integer)===>20000
Введите значение D (Integer)===>1
ПАСКАЛЬ:    x=20
АССЕМБЛЕР: x=-12
Повторим? (y/n)
```

Нам ОЧЕНЬ повезло — мы ПОЛУЧИЛИ ошибку!

2. **Описание ошибки.** Чтобы понять, что произошло, нужно об этой ошибке кому-то рассказать (*"одна голова хорошо, а полторы головы лучше"*, — Пудовкина Л.Ф., моя коллега по работе), даже можно своему домашнему любимцу. *"Оказывается, моя кошка прекрасно разбирается в отладке"*, — пишет в своей книге Д. Роббинс. Я то же самое могу сказать о своем сиамском коте Байте Пинкертоне (Пиньке), который постоянно сидит на моих коленях за компьютером и ОЧЕНЬ мне помогает (телепатически, а иногда и лапой). Итак, что же произошло в нашем случае? Какой из модулей дал правильный результат, и какой он должен быть? Прикинем быстренько результат. Знак минус НИКАК НЕ может быть, а вот выражение в числителе должно получиться достаточно большим (за пределами 16 бит). Что-то здесь есть... Возможно, в этом и заключается ошибка? Пинька, как ты думаешь? Тебе жарко, мне тоже, но книгу писать надо и ошибку ловить тоже надо... Пойдем дальше...
3. **Всегда вините в ошибках себя.** Начинающие программисты (а особенно студенты при сдаче лабораторных работ) стараются всю вину свалить на что угодно:
   - на операционную систему (такое бывает, но ОЧЕНЬ редко и, в основном, с СИСТЕМНЫМИ программами!);
   - на версию компилятора (это бывает чаще, и в нашем случае, например, **Object Pascal (Delphi)** на ЭТИХ данных считал бы НОРМАЛЬНО! И мы уже знаем, почему, не правда ли? — см. Приложение 4);
   - отлаженный файл программы куда-то задевался, а он (студент) сдает старую версию и проч.
4. **Разделяйте и властвуйте.** Описав ошибку, надо постараться определить, в чем она заключается и где может находиться. Посчитаем наше выражение: числитель = 2*20000-1 = 40000 —1 = 39999. Знаменатель = 1000+999=1999. Теперь разделим 39999/1999 = 20. Все ясно — ошибка в Ассемблере. Почему? Посмотрим на исходный код числителя:

```
; ———      числитель
           mov      ax, 2       ; ax <=== 2
           Imul     c           ; <dx:ax>=2*c
           sub      ax,d        ; <ax>=<ax>-d
```

Причина найдена! При вычитании **2*c-d** мы потеряли содержимое регистра **<dx>**, полученного в результате умножения **2*c**. В данной ситуации возможны два выхода. Можно код в Ассемблере так и оставить, но тогда в Паскале нужно сделать проверку попадания **числителя** в нужный диапазон:

```
templ := 2.* c - d;
if (templ>=Min)and(templ<=Max) then ....
```

Но этот вариант НЕОПРАВДАННО сузит область допустимых значений для **c** и **d**, и нас это НЕ устраивает. Поэтому мы его отбрасываем и переходим к следующему варианту — надо исправить код на Ассемблере.

5. **Творческий подход.** К сожалению, не всегда удается так быстро найти и локализовать ошибку (кстати, в нашей программе есть еще одна ошибка, которая пока никак НЕ проявилась, она достаточно хитрая!). Иногда, в ОЧЕНЬ сложных случаях лучше отложить решение проблемы на один-два дня или пойти выспаться, а подсознание-то все равно работает (вспомните Д.И. Менделеева!). А я однажды получила решение проблемы при просмотре советского ужастика "Вий".

6. **Необходимые инструменты.** Мы с вами пока будем использовать, как уже упоминалось, интегрированные отладчики. К книге Д. Роббинса прилагается компакт-диск, в который, кроме исходных текстов примеров, включен графический визуальный отладчик (очень похожий на прекрасный отладчик, встроенный в **Visual C++ 6.0**) и восемь утилит, ОЧЕНЬ облегчающих процесс отладки СЕРЬЕЗНЫХ программ, например, приложений для Windows.

7. **Начало серьезной отладки.** В этом случае, конечно, нужно использовать МОЩНЫЙ отладчик и задействовать его дополнительные функции, чтобы он выполнял возможно большую часть тяжелой работы. Нам с вами, слава Богу, такая отладка пока НЕ нужна... Кроме того, она совсем НЕ для начинающих... И не по теме нашей книги.

8. **Убедитесь, что ошибка устранена.** А вот это как раз по нашей части — переписываем код вычисления числителя с учетом нашей гипотезы о регистре **DX**:

```
; ———— числитель
        mov     ax, 2       ; ax <=== 2
        Imul    c           ; <dx:ax>=2*c
; готовимся к расширению d (Word) в DWord
        mov     cx, ax      ; СОХРАНЕНИЕ регистров AX,
        mov     SI, dx      ;                      DX
; делаем расширение d (Word) ===> DWord
        mov     ax, d
        CWD                 ; <dx:ax> <=== <ax>
;     Теперь можно и вычитать!
        sub     cx,ax       ; <cx>=<cx>-<ax> мл. часть
        sbb     SI,dx       ; <SI>=<SI>-<dx> ст.часть
;   готовимся к операции ДЕЛЕНИЯ:
        xchg    cx,ax       ; обмен содержимым регистров
        xchg    SI,dx
```

Да, код ОЧЕНЬ изменился, не правда ли?
Теперь надо все перекомпилировать и проверить правильность нашей гипотезы.

```
———————— Test #2 ————————————
 Вычислить:        x = (2*c-d)/(a+b); a,b,c,d,x : Integer;
Введите значение A (Integer)===>1000
Введите значение B (Integer)===>999
Введите значение C (Integer)===>20000
Введите значение D (Integer)===>1

ПАСКАЛЬ:    x=20
АССЕМБЛЕР:  x=20

Повторим? (y/n)
 Вычислить:        x = (2*c-d)/(a+b); a,b,c,d,x : Integer;
Введите значение A (Integer)===>1111
```

```
Введите значение B (Integer)===>
-1111
Введите значение C (Integer)===>111
Введите значение D (Integer)===>1
!!!!! Знаменатель равен НУЛЮ — вычисления невозможны !!!!
Введите значение A (Integer)===>11111
Введите значение B (Integer)===>-1111
Введите значение C (Integer)===>30000
Введите значение D (Integer)===>0

ПАСКАЛЬ:    x=6
АССЕМБЛЕР: x=6

Повторим? (y/n)
 Вычислить:       x = (2*c-d)/(a+b); a,b,c,d,x : Integer;
Введите значение A (Integer)===>10000
Введите значение B (Integer)===>999
Введите значение C (Integer)===>10000
Введите значение D (Integer)===>1

ПАСКАЛЬ:    x=1
АССЕМБЛЕР: x=1

Повторим? (y/n)
 Вычислить:       x = (2*c-d)/(a+b); a,b,c,d,x : Integer;
Введите значение A (Integer)===>10
Введите значение B (Integer)===>-9
Введите значение C (Integer)===>20000
Введите значение D (Integer)===>0
Результат выходит за диапазон [-32768..32767]!!!!
 4.00000000000000E+0004
Повторите ввод.
Введите значение A (Integer)===>20000
Введите значение B (Integer)===>20000
Введите значение C (Integer)===>30000
Введите значение D (Integer)===>0

ПАСКАЛЬ:    x=-2
АССЕМБЛЕР: x=-2

Повторим? (y/n)
 Вычислить:       x = (2*c-d)/(a+b); a,b,c,d,x : Integer;
Введите значение A (Integer)===>20000
Введите значение B (Integer)===>20000
Введите значение C (Integer)===>10000
Введите значение D (Integer)===>0

ПАСКАЛЬ:    x=0
АССЕМБЛЕР: x=0

Повторим? (y/n)
 Вычислить:       x = (2*c-d)/(a+b); a,b,c,d,x : Integer;
Введите значение A (Integer)===>100
Введите значение B (Integer)===>-99
Введите значение C (Integer)===>20000
Введите значение D (Integer)===>0
Результат выходит за диапазон [-32768..32767]!!!!
 4.00000000000000E+0004
Повторите ввод.
```

```
Введите значение A (Integer)===>20000
Введите значение B (Integer)===>20000
Введите значение C (Integer)===>40000
Вводимое значение выходит за диапазон [-32768..32767]!!!!
Повторите ввод.
Введите значение C (Integer)===>32700
Введите значение D (Integer)===>0

ПАСКАЛЬ:     x=-2 ??????????
АССЕМБЛЕР:   x=-2 ??????????

Повторим? (y/n)
```

Вот и проявилась наша хитрая ошибка. Последний результат НЕВЕРНЫЙ — должна получиться **ЕДИНИЦА:**

(2*32700-0)/(20000+20000) = 65400/40000 = 1,635.

Теперь у нас вышел за диапазон знаменатель (посмотрите внимательно на тест — знаменатель и раньше выходил за диапазон, но ошибка НЕ проявлялась!), а Ассемблер и Паскаль на него отреагировали ОДИНАКОВО и дали НЕВЕРНЫЙ результат! Что же делать? Предоставляю это вам, дорогие мои читатели! Учитесь, пробуйте, набирайтесь опыта. **Ошибка проявилась — исправить ее — теперь ваша задача!** Это и будет тест на проверку ПОНИМАНИЯ вами прочитанного. Удачи вам!

9. **Учитесь и распространяйте опыт.** Д. Роббинс советует тщательно записывать все свои шаги по исправлению ошибок в специальную тетрадь (я тоже так делаю!), чтобы, во-первых, НЕ повторяться, во-вторых, накапливать опыт и делиться им с другими — так люди пишут свои хорошие, *грамотные* программы, а потом и книги...

6.3.3. Краткий обзор интегрированных отладчиков

Ну а мы вернемся к изучению Ассемблера и к имеющимся в нашем распоряжении инструментам отладки. Самый простой и естественный — это интегрированный отладчик. Он видит и модуль на алгоритмическом языке, и модуль на Ассемблере (если вы прописали все пути — в крайнем случае, нужно компилятор с Ассемблера переписать в свою текущую директорию). С интегрированным отладчиком среды разработки **Turbo Pascal-7.0x** мы уже немного познакомились в п. 5.1.1.2. Он, конечно, поддерживает ТОЛЬКО 16-разрядные приложения.

Рассмотрим еще один отладчик для 16-разрядных приложений, полученных путем стыковки **Cpp+Assembler** (на базе **Borland C++ 3.1 и Turbo Assembler 3.1**). Идейно он практически ничем от интегрированного отладчика среды разработки **Turbo Pascal-7.0x** НЕ отличается.

1. Нужно создать **проект** из стыкуемых модулей **Cpp+Assembler**. Редактировать эти модули можно тоже в интегрированной среде.
2. Занести в EXE-файл отладочную информацию — **Options | Compiler | Advanced Code generation:**
   - **Options — Debug info in OBJs.**

3. Подключить отладчик — **Options | Debugger**:
   - **Source Debugging — On**
   - **Display Swapping — Smart**
4. Установить контрольные точки, например, так: **F4 — Go to cursor**.
5. Подключить окно CPU: **Window | Register**.
6. Запустить отладочный режим: **F7 — trace** (с трассировкой подпрограмм), **F8 — Step** (БЕЗ трассировки подпрограмм).

В качестве тестового примера можно использовать пример 6.2а, только надо перекодировать кириллицу из кодировки **Win** в кодировку **DOS** — получим пример 6.5. Пример 6.2b для наших целей НЕ подходит, т. к использует современные средства С++, которые данная версия компилятора (**ВС++ 3.1**) НЕ поддерживает.

Интегрированная среда **Borland C++ 5.02** отладку 16-разрядных приложений уже НЕ поддерживает. Можно сделать 32-разрядное приложение со *встроенным Ассемблером* (см. пример 6.1, п. 6.2.2.1) и запустить отладчик, но, по мнению автора, сделать это лучше в среде программирования **Microsoft Visual C++ 6.0** (см. п. 6.2.2.2). Он содержит прекрасный визуальный отладчик с интуитивно понятным интерфейсом (в виде панели). Кроме того, видны ВСЕ возможные регистры, можно подключить окно дизассемблера и полюбопытствовать, как транслируется код С++ в Ассемблер. Есть много всяких приятных и полезных для программиста настроек.

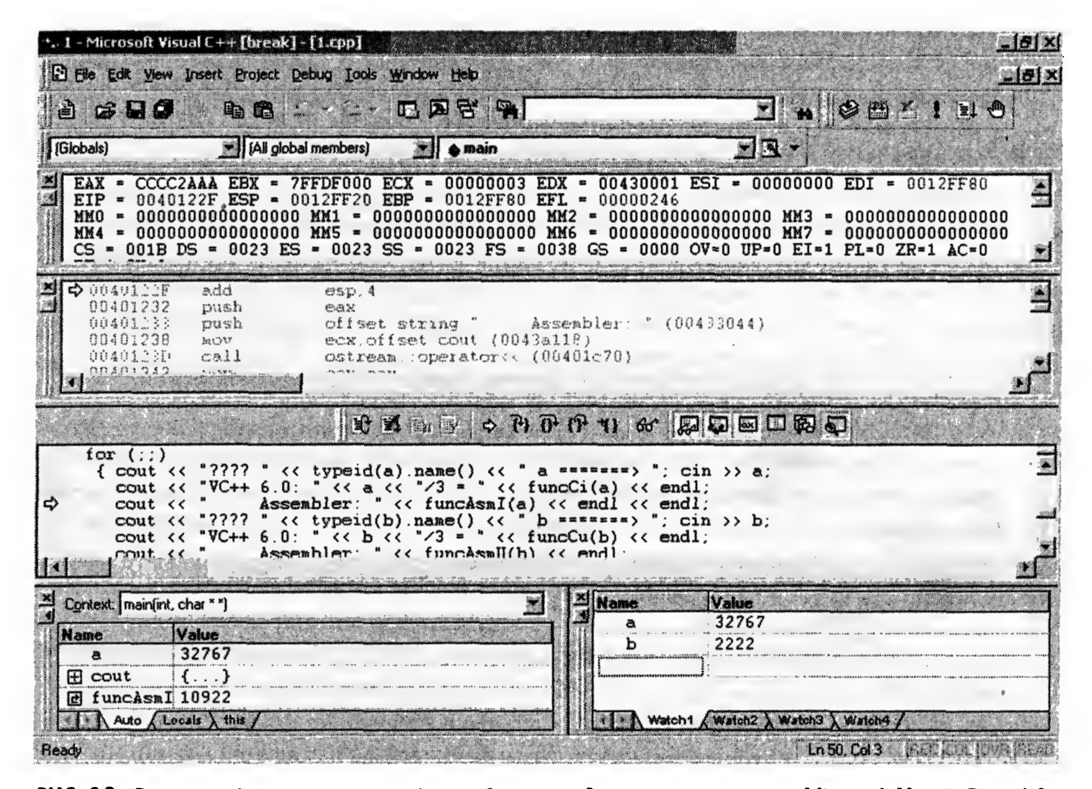
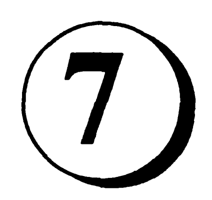


РИС. 6.3. *Визуальный интегрированный отладчик в среде программирования Microsoft Visual C++ 6.0.*



Основные команды работы с битами для IBM PC XT

Многие вещи нам непонятны не потому, что наши понятия слабы; но потому, что сии вещи не входят в круг наших понятий.

Отыщи всему начало, и ты многое поймешь.

Козьма Прутков

Это — родные операции компьютера, поэтому они выполняются ОЧЕНЬ быстро, но, к сожалению, не всегда понятны человеку. Попробуем разобраться в командах обработки битов. Раз они так важны компьютеру, значит, они должны стать родными и человеку-программисту.

Различают собственно ЛОГИЧЕСКИЕ операции и СДВИГИ. Они производятся над битами байта или слова, поэтому еще называются *побитовыми* (*поразрядными*). Эти операции есть во всех серьезных алгоритмических языках, в том числе и в наших базовых — Паскаль и C/C++. В этих языках они определены для операндов ЦЕЛОГО и/или ЛОГИЧЕСКОГО типов данных.

7.1. Логические команды

Теоретической базой для логических операций является *булева алгебра*, которая впервые была исследована Джоффуром Булем (1815-1864). Она базируется на высказываниях. *Высказывание* — это законченное предложение, о котором можно **определенно** сказать, что его содержание *истинно* или *ложно*. Операции булевой алгебры определены для таких высказываний и в качестве результата дают *таблицу истинности* — см. табл. 7.1.

Исчисление высказываний прекрасно сочетается с принципами работы компьютера и основными методами его программирования [Л2-3]. Мы с вами уже знаем, что минимальной единицей информации в компьютере является **бит**, который имеет два состояния: **0 (ложь)** и **1 (истина)**. Есть даже языки *логического* программирования (например, ПРОЛОГ), которые реализуют исчисление высказываний, и автору этих строк посчастливилось довольно долго с ПРОЛОГОМ работать...

Таблица 7.1. Основные операции булевой алгебры (математической логики).

Операнды					Операции	Название операции в математической логике
a	0	0	1	1		
b	0	1	0	1		
Результат логической операции (**таблица истинности**)	1	1	0	0	**NOT** a	Отрицание (инверсия, NOT)
	0	0	0	1	a **AND** b	Конъюнкция, логическое умножение (AND)
	0	1	1	0	a **XOR** b	Исключающее ИЛИ (сложение по модулю 2, XOR)
	0	1	1	1	a **OR** b	Дизъюнкция, логическое сложение (OR)

Логические операции в алгоритмических языках Паскаль и C/C++ определены для операндов ЦЕЛОГО и/или ЛОГИЧЕСКОГО типов данных и используются чаще всего в логических условиях. Кстати, в языке C++ (идейно ОЧЕНЬ близкого к Ассемблеру) только совсем недавно появился логический тип данных **bool** и даже не все компиляторы C++ его ОДИНАКОВО поддерживают.

Таблица 7.2. Логические операции над битами в алгоритмических языках Паскаль и C/C++.

Логические операции	Обозначение в алгоритмических языках		Примеры применения в алгоритмических языках	
	C/C++	Паскаль	C/C++	Паскаль
Побитовое И (AND)	&	**AND**	Alpha & 512	Alpha **AND** 512
Побитовое ИЛИ (OR)	|	**OR**	Beta | mask	Beta **or** mask
Побитовое сложение по модулю 2 (XOR)	^	**XOR**	h ^ Beta	h **xor** Beta
Побитовое отрицание (NOT)	~	**NOT**	~ k	**Not** k

Для Ассемблера, как мы знаем, понятие ТИП данных (в том смысле, как это понимается в алгоритмических языках) НЕ существует. Поэтому эти побитовые операции применяются (согласно правилам математической логики) к каждому биту байта или слова ОТДЕЛЬНО, начиная с младшего бита 0 и заканчивая старшим битом.

В Ассемблере логические команды реализуют четыре основные операции математической логики:

- **Логическое отрицание (NOT).**
- **Логическое умножение (AND).**
- **Логическое сложение (OR).**
- **Сложение по модулю 2 (XOR).**

Логическое отрицание — это операция ОДНОМЕСТНАЯ (ей нужен только ОДИН операнд), все остальные — ДВУХМЕСТНЫЕ.

Таблица 7.3. Логические команды в Ассемблере.

Синтаксис	Логика работы команды	Биты регистра флагов				
		OF	SF	ZF	PF	CF
AND *Приемник, Источник*	*<Приемник>* = *<Приемник>* **AND** *<Источник>*	0	+	+	+	0
TEST *Приемник, Источник*	*<Приемник>* **AND** *<Источник>*	0	+	+	+	0
OR *Приемник, Источник*	*<Приемник>* = *<Приемник>* **OR** *<Источник>*	0	+	+	+	0
XOR *Приемник, Источник*	*<Приемник>* = *<Приемник>* **XOR** *<Источник>*	0	+	+	+	0
NOT *Приемник*	*<Приемник>* = **NOT** *<Приемник>*	-	-	-	-	-

Для двухместных команд (в соответствии с общепринятыми в Ассемблере правилами) происходит установка (или очистка) соответствующих битов регистра флагов (+). Возможные сочетания операндов такие же, как и приведенные в табл. 5.1.

Команда **NOT** содержимое регистра флагов НЕ меняет.

<Источник> в логических командах с двумя операндами обычно именуется *маска*. Отсюда и термин *маскирование разрядов*. Значение маски обычно берут или в двоичном, или в шестнадцатеричном виде. Оно означает, что в соответствующем бите установлена единица (см. табл. 7.4).

Таблица 7.4. Шестнадцатеричные маски

Номер бита	Значение маски (HEX)	Номер бита	Значение маски (HEX)
0	0001	8	0100
1	0002	9	0200
2	0004	10	0400
3	0008	11	0800
4	0010	12	1000
5	0020	13	2000
6	0040	14	4000

7.1.1. Использование команд логического умножения AND и TEST

Результат поразрядного логического умножения дает значение **истина (1)** только в **одном случае**: если соответствующие разряды *приемника* и *маски* тоже установлены в 1 — см. табл. 7.1. Поэтому эта операция обычно применяется для **проверки установки** нужных нам разрядов в 1.

Разница между командами **AND** и **TEST** заключается в том, что команда **TEST** используется для организации *логического сравнения операндов* (по аналогии с командой *арифметического сравнения операндов* **CMP**) с последующим *условным* переходом, поэтому она содержимое *приемника* НЕ меняет.

 **ПРИМЕР 7.1.**

Пусть в процессе каких-то вычислений сформировалась ЦЕЛОЧИСЛЕННАЯ переменная x длиной 16 бит. Проверить содержимое битов **1, 2, 10, 15** и выдать соответствующую информацию на дисплей. Будем решать нашу задачу на языке C/C++ и на встроенном Ассемблере (**BC++ 5.02**) — по сути дела эти две реализации абсолютно идентичны. Значение переменной x сначала для удобства анализа зададим в шестнадцатеричном виде в программе, а затем будем вводить ЛЮБЫЕ значения и делать выводы.

Воспользовавшись таблицей 7.4, сформируем маску проверки как результат сложения составляющих масок:

Таблица 7.5. Формирование маски проверки.

Необходимые биты	**Маски (HEX-код)**
Бит 1	0002
Бит 2	0004
Бит 10	0400
Бит 15	8000
Результирующая маска	8406

Теперь можно применять операцию логического умножения и проверять все нужные нам биты сразу или по одному.

Исходный текст программы bits.cpp

```
/* bits.cpp    Borland C++ 4.5 (5.02)      EasyWin
     (c) Copyright 1998-2001 by Голубь Н.Г.
        Лекционный пример 7.1 – вариант 1.
Пусть в процессе каких-то вычислений сформировались ЦЕЛОЧИСЛЕННАЯ переменная x длиной 16 бит. Проверить содержимое битов 1, 2, 10, 15 и выдать информацию на дисплей.
       Операции над БИТАМИ

*/
#include <iostream.h>
#include <conio.h>
#include <typeinfo.h>

void bitAND(int&);
int x=0xFA01, k;

void main()
{ int t=0;
  bitAND(t);
  for (;;)
   { clrscr();
    cout    << "\nВводите значение переменной типа " << typeid(x).name()
            << " x ======> "; cin >> x;
    bitAND(t);
   };
}
```

```
void bitAND(int& t)
{cout     << "\n===================== Test #" << ++t
          << " ========================";
  cout    << "\n—————————— Начальное значение: ——————————\n"
          << "x = " << hex << x <<"(hex); "<< dec << x << "(dec)\n";
  k=x&0x8406;
  cout    << "Полученное значение:\nk = x&8406 = " << hex << k
          <<"(hex);  "<< dec << k << "(dec)" << endl;
  cout    << "\n Проверим по битам:\n Бит 1:  ";
  asm { mov ax,x
        AND ax,0x0002  ; // бит 1
        mov k,ax
        }
  cout    << "Полученное значение:\nk = x&0x0002 = " << hex << k
          <<"(hex);  "<< dec << k << "(dec)" << endl;
  cout    << "\n Проверим по битам:\n Бит 2:  ";
  asm { mov ax,x
        AND ax,0x0004  ; // бит 2
        mov k,ax
        }
  cout    << "Полученное значение:\nk = x&0x0004 = " << hex << k
          <<"(hex);  "<< dec << k << "(dec)" << endl;
  cout    << "\n Проверим по битам:\n Бит 10:  ";
  asm { mov ax,x
        AND ax,0x0400  ; // бит 10
        mov k,ax
        }
  cout    << "Полученное значение:\nk = x&0x0400 = " << hex << k
          <<"(hex);  "<< dec << k << "(dec)" << endl;
  cout    << "\n Проверим по битам:\n Бит 15 — знак числа!!!:  ";
  asm { mov ax,x
        AND ax,0x8000  ; // бит 15 — знак!!!
        mov k,ax
        }
  cout    << "Полученное значение:\nk = x&0x8000 = " << hex << k
          <<"(hex);  "<< dec << k << "(dec)" << endl;
  cout    << "\nПродолжим?              ALT-F4 — выход....\n";
  getch();
}
```

Результат работы этой программы:

```
===================== Test #1 ========================
————————— Начальное значение: —————————-
x = ffffa01(hex); -1535(dec)
Полученное значение:
k = x&8406 = ffff8000(hex);   -32768(dec)

Проверим по битам:
Бит 1:  Полученное значение:
k = x&0x0002 = 0(hex);  0(dec)

Проверим по битам:
Бит 2:  Полученное значение:
k = x&0x0004 = 0(hex);  0(dec)
```

```
Проверим по битам:
Бит 10:  Полученное значение:
k = x&0x0400 = 0(hex);  0(dec)
Проверим по битам:
Бит 15 — знак числа!!!:  Полученное значение:
k = x&0x8000 = ffff8000(hex);  -32768(dec)

Продолжим?                    ALT-F4 — выход....
Вводите значение переменной типа int x ======> -12345

==================== Test #2 =======================

———————— Начальное значение: ——————————-
x = ffffcfc7(hex); -12345(dec)
Полученное значение:
k = x&8406 = ffff8406(hex);  -31738(dec)

Проверим по битам:
Бит 1:  Полученное значение:
k = x&0x0002 = 2(hex);  2(dec)

Проверим по битам:
Бит 2:  Полученное значение:
k = x&0x0004 = 4(hex);  4(dec)

Проверим по битам:
Бит 10:  Полученное значение:
k = x&0x0400 = 400(hex);  1024(dec)
Проверим по битам:
Бит 15 — знак числа!!!:  Полученное значение:
k = x&0x8000 = ffff8000(hex);  -32768(dec)

Продолжим?                    ALT-F4 — выход....
Вводите значение переменной типа int x ======> 12345

==================== Test #3 =======================

———————— Начальное значение: ——————————-
x = 3039(hex); 12345(dec)
Полученное значение:
k = x&8406 = 0(hex);  0(dec)

Проверим по битам:
Бит 1:  Полученное значение:
k = x&0x0002 = 0(hex);  0(dec)

Проверим по битам:
Бит 2:  Полученное значение:
k = x&0x0004 = 0(hex);  0(dec)

Проверим по битам:
Бит 10:  Полученное значение:
k = x&0x0400 = 0(hex);  0(dec)

Проверим по битам:
Бит 15 — знак числа!!!:  Полученное значение:
k = x&0x8000 = 0(hex);  0(dec)

Продолжим?                    ALT-F4 — выход....
```

ОБРАТИТЕ ВНИМАНИЕ НА ТЕСТ #2.

Маска **0x8406** и результат операции умножения получились ОДИНАКОВЫМИ! Это значит, что в двоичном виде число –12345 имеет единицы в маскируемых битах 1, 2, 10 и 15. Это же подтверждает и проверка по отдельным битам.

Можете вводить свои значения и поэкспериментировать, чтобы понять, что же происходит... А лучше всего вручную расписать число в двоичном виде и применить к нему операцию **AND**. Как видите, логические операции достаточно специфичны. Они НЕ имеют НИЧЕГО общего (в смысле десятичного результата) с привычными для человека арифметическими операциями. В данном примере десятичный вывод, возможно, даже избыточен, т.к. способен запутать ситуацию. Но в то же время он показывает, что ИНТЕРПРЕТАЦИЯ результата — дело человека, а НЕ компьютера (для него родными являются ТОЛЬКО двоичные числа!). Наверное, поэтому для начинающих программистов, которые хотят найти аналогию с чем-то уже знакомым, логические операции поначалу и представляют такую сложность. НЕ пугайтесь. **Эти операции имеют смысл при работе с БИТАМИ** и только! Они, например, ОЧЕНЬ широко используются при организации графического вывода для получения самых разных интересных эффектов.

7.1.2. Использование команды логического сложения OR

Результат поразрядного логического сложения дает значение **ложь (0)** только в **одном случае**: если соответствующие разряды *приемника* и *маски* тоже равны нулю — см. табл. 7.1. Поэтому эта операция обычно применяется для **установки** нужных нам разрядов в 1.

Теперь продолжим наш пример 7.1 и сделаем установку битов 13, 9 и 7 в единицу. Рассуждая, аналогично предыдущему, получим шестнадцатеричную маску **0x2280**. Внесем необходимые изменения в программу **bits.cpp**.

7.1.3. Использование команды сложения по модулю 2 — XOR

Результат поразрядного логического сложения по модулю 2 дает значение **ложь (0)** в **двух случаях**: если соответствующие разряды *приемника* и *маски* **одинаковые**, и значение **истина (1)** тоже в **двух случаях**: если соответствующие разряды *приемника* и *маски* **разные** — см. табл. 7.1. Поэтому эта операция обычно применяется для **обнуления** нужных нам разрядов, если они установлены в 1.

Продолжим наш пример 7.1. Проверить содержимое битов 14,15. Если они равны 1, следует обнулить их. Рассуждая, аналогично предыдущему, получим шестнадцатеричную маску 0x4000+0x8000 = **0xC000**. Опять внесем необходимые изменения в программу **bits.cpp**.

7.1.4. Команда логического отрицания NOT

Это достаточно простая команда. Продолжая наш пример 7.1, сделаем две инверсии: на языке C++ и встроенном Ассемблере. В результате должны получить введенное значение переменной **x**.

Окончательная версия программы — файл bits.cpp

```
/* bits.cpp    Borland C++ 4.5 (5.02)     EasyWin
                    (c) Copyright 1998-2001 by Голубь Н.Г.
        Лекционный пример 7.1 — окончательный вариант.
    Пусть в процессе каких-то вычислений сформировалась ЦЕЛОЧИСЛЕННАЯ переменная
                        x длиной 16 бит (мы ее вводим).
    1) Проверить содержимое битов 1, 2, 10, 15 (AND).
    2) Установить в 1 биты 7, 9, 13 (OR).
    3) Проверить содержимое битов 14,15. Если они равны 1, обнулить их (XOR).
    4) Сделать двойную инверсию: на языке C++ и встроенном Ассемблере
```

```
    Логические операции над БИТАМИ (AND, OR, XOR, NOT)
*/

#include <iostream.h>
#include <conio.h>
#include <typeinfo.h>

void bitAND(int&);
void bitOR(int&);
void bitXOR(int&);
void bitNOT(int&);

int x=0xFA01,k;

void main()
{int t=0;
  char c;
  for (;;)
     {clrscr();
      cout << "\nВводите значение переменной типа " << typeid(x).name()
           << " x ======> "; cin >> x;
      cout << "\nВаш выбор логической операции: ~ (NOT), & (AND), | (OR), ^ (XOR) =>  ";
      cin >> c;
       switch (c)
           {case "&": bitAND(t); break;
            case "|":  bitOR(t);  break;
            case "^": bitXOR(t); break;
            case "~": bitNOT(t); break;
            default: cout << "\nВы ввели НЕВЕРНЫЙ символ!!!!!!";
           }
      cout << "\nПродолжим?                ALT-F4 — выход....\n";
      getch();
     };
}

void bitAND(int& t)
{cout     << "\n================= Test #" << ++t << " (AND) ==================";
 cout     << "\n————————————— Начальное значение: ———————————————\n"
          << "x = " << hex << x <<"(hex); "<< dec << x << "(dec)\n";
 k=x&0x8406;
 cout     << "Полученное значение:\nk = x&8406 = " << hex << k
          << "(hex);  "<< dec << k << "(dec)" << endl;
 cout     << "\n Проверим по битам:\n Бит 1:  ";
 asm      { mov ax,x
           AND ax,0x0002 ; // бит 1
           mov k,ax
          }
  cout    << "Полученное значение:\nk = x&0x0002 = " << hex << k
          << "(hex);  "<< dec << k << "(dec)" << endl;
  cout    << "\n Проверим по битам:\n Бит 2:  ";
  asm { mov ax,x
        AND ax,0x0004 ; // бит 2
        mov k,ax
      }
  cout    << "Полученное значение:\nk = x&0x0004 = " << hex << k
          << "(hex);  "<< dec << k << "(dec)" << endl;
  cout    << "\n Проверим по битам:\n Бит 10:  ";
```

```
  asm { mov ax,x
          AND ax,0x0400 ; // бит 10
          mov k,ax
        }
  cout     << "Полученное значение:\nk = x&0x0400 = " << hex << k
           << "(hex);  "<< dec << k << "(dec)" << endl;
  cout     << "\n Проверим по битам:\n Бит 15 — знак числа!!!:  ";
  asm { mov ax,x
          AND ax,0x8000 ; // бит 15 — знак!!!
          mov k,ax
      }
  cout     << "Полученное значение:\nk = x&0x8000 = " << hex << k
           << "(hex);  "<< dec << k << "(dec)" << endl;
}
void bitOR(int& t)
{cout      << "\n================= Test #" << ++t << " (OR) =================";
 cout      << "\n—————————— Начальное значение: ——————————\n"
           << "x = " << hex << x <<"(hex); "<< dec << x << "(dec)\n";
 k=x|0x2280;
 cout      << "Полученное значение:\nk = x|2280 = " << hex << k
           << "(hex);  "<< dec << k << "(dec)" << endl;
 cout      << "\n Проверим по битам:\n Бит 7:  ";
 asm { mov ax,x
          OR  ax,0x0080 ; // бит 7
          mov k,ax
       }
  cout     << "Полученное значение:\nk = x|0x0080 = " << hex << k
           << "(hex);  "<< dec << k << "(dec)" << endl;
  cout     << "\n Проверим по битам:\n Бит 9:  ";
  asm { mov ax,x
          OR  ax,0x0200 ; // бит 9
          mov k,ax
       }
  cout     << "Полученное значение:\nk = x|0x0200 = " << hex << k
           << "(hex);  "<< dec << k << "(dec)" << endl;
  cout     << "\n Проверим по битам:\n Бит 13:  ";
  asm {  mov ax,x
          OR  ax,0x2000 ; // бит 13
          mov k,ax
       }
  cout     << "Полученное значение:\nk = x|0x2000 = " << hex << k
           << "(hex);  "<< dec << k << "(dec)" << endl;
}
void bitXOR(int& t)
{cout      << "\n================= Test #" << ++t << " (XOR) ================";
 cout      << "\n—————————— Начальное значение: ——————————\n"
           << "x = " << hex << x <<"(hex); "<< dec << x << "(dec)\n";
 k=x^0xC000;
 cout      << "Полученное значение:\nk = x^C000 = " << hex << k
           << "(hex);  "<< dec << k << "(dec)" << endl;
 cout      << "\n Проверим по битам:\n Бит 14:  ";
 asm { mov ax,x
          XOR ax,0x4000 ; // бит 14
          mov k,ax
      }
```

```
    cout     << "Полученное значение:\nk = x^0x4000 = " << hex << k
             << "(hex);  "<< dec << k << "(dec)" << endl;
    cout     << "\n Проверим по битам:\n Бит 15:  ";
    asm  { mov ax,x
           XOR ax,0x8000 ; // бит 15
           mov k,ax
         }
    cout     << "Полученное значение:\nk = x^0x8000 = " << hex << k
             << "(hex);  "<< dec << k << "(dec)" << endl;
}
void bitNOT(int& t)
{ cout       << "\n================= Test #" << ++t << " (NOT) ================";
  cout       << "\n————————— Начальное значение: —————————\n"
             << "x = " << hex << x <<"(hex); "<< dec << x << "(dec)\n";
  k=~x;
  cout       << "Полученное значение:\nk = ~x = " << hex << k
             << "(hex);  "<< dec << k << "(dec)" << endl;
  asm  { mov ax,k
         not ax
         mov k,ax
       }
  cout       << "Вернемся назад:\nk = NOT k = " << hex << k
             << "(hex);  "<< dec << k << "(dec)" << endl;
}
```

Тестовые примеры:

```
Вводите значение переменной типа int x ======> -11223

Ваш выбор логической операции: ~ (NOT), & (AND), | (OR), ^ (XOR) ==>

===================== Test #1 (NOT) =====================

———————— Начальное значение: ——————————
x = ffffd429(hex); -11223(dec)
Полученное значение:
k = ~x = 2bd6(hex);  11222(dec)
Вернемся назад:
k = NOT k = ffffd429(hex);  -11223(dec)

Продолжим?                  ALT-F4 — выход....

Вводите значение переменной типа int x ======> 32212

Ваш выбор логической операции: ~ (NOT), & (AND), | (OR), ^ (XOR) ==>

===================== Test #2 (AND) =====================
———————— Начальное значение: ——————————
x = 7dd4(hex); 32212(dec)
Полученное значение:
k = x&8406 = 404(hex);  1028(dec)

Проверим по битам:
Бит 1:  Полученное значение:
k = x&0x0002 = 0(hex);  0(dec)

Проверим по битам:
Бит 2:  Полученное значение:
k = x&0x0004 = 4(hex);  4(dec)
```

```
Проверим по битам:
Бит 10: Полученное значение:
k = x&0x0400 = 400(hex);  1024(dec)

Проверим по битам:
 Бит 15 — знак числа!!!: Полученное значение:
k = x&0x8000 = 0(hex);  0(dec)

Продолжим?                  ALT-F4 — выход....

Вводите значение переменной типа int x ======> -22211

Ваш выбор логической операции: ~ (NOT), & (AND), | (OR), ^ (XOR) ==>  |

===================== Test #3 (OR) =====================

———————— Начальное значение: ————————
x = ffffa93d(hex); -22211(dec)
Полученное значение:
k = x|2280 = ffffabbd(hex);  -21571(dec)

Проверим по битам:
Бит 7: Полученное значение:
k = x|0x0080 = ffffa9bd(hex);  -22083(dec)

Проверим по битам:
Бит 9: Полученное значение:
k = x|0x0200 = ffffab3d(hex);  -21699(dec)

Проверим по битам:
Бит 13: Полученное значение:
k = x|0x2000 = ffffa93d(hex);  -22211(dec)

Продолжим?                  ALT-F4 — выход....
Вводите значение переменной типа int x ======> -4325
Ваш выбор логической операции: ~ (NOT), & (AND), | (OR), ^ (XOR) ==>  ^

===================== Test #4 (XOR) =====================
———————— Начальное значение: ————————
x = ffffef1b(hex); -4325(dec)
Полученное значение:
k = x^C000 = 2f1b(hex);  12059(dec)

Проверим по битам:
Бит 14: Полученное значение:
k = x^0x4000 = ffffaf1b(hex);  -20709(dec)

Проверим по битам:
Бит 15: Полученное значение:
k = x^0x8000 = 6f1b(hex);  28443(dec)

Продолжим?                  ALT-F4 — выход....
```

Проанализируйте внимательно полученные результаты, сделайте свои варианты ввода значения числа x. Поработайте с битами вручную, и основные принципы работы компьютера станут вам понятнее и ближе.

7.2. Команды сдвига

Это — другая разновидность побитовых команд, которые перемещают (сдвигают) биты в поле операнда *Приемник*, либо влево (**Left**), либо вправо (**Right**) на определенное число разрядов — это значение хранится в *Источнике*. Поэтому в мнемонике этих команд обязательно присутствует буква **L** или **R**. Все команды сдвига состоят из двух операндов.

Синтаксис:

Мнемокод_команды_сдвига *Приемник, Источник*

Команды сдвига по принципу действия разделяются на линейные (арифметический и логический сдвиг) и циклические сдвиги.

Таблица 7.6. Возможные сочетания операндов для команд сдвига.

Приемник	Источник (счетчик)	Пояснение
R8	1	Сдвиг на один разряд
R16	1	
Mem8	1	
Mem16	1	
R8	CL	Количество сдвигаемых разрядов находится в регистре < CL > = [0..31]
R16	CL	
Mem8	CL	
Mem16	CL	
R8	Im8	Количество сдвигаемых разрядов находится в константе Im8 = [2..31]. **Операция определена, начиная с i286.**
R16	Im8	
Mem8	Im8	
Mem16	Im8	

Линейные команды сдвига добавляют НОЛЬ либо в младший (нулевой) разряд (при сдвиге ВЛЕВО — L), либо — в старший (при сдвиге ВПРАВО — R). Причем старший разряд при сдвиге ВПРАВО, естественно, РАЗЛИЧАЕТСЯ для знаковых и беззнаковых данных. Сдвиг для знаковых данных называется АРИФМЕТИЧЕСКИМ, а для беззнаковых — ЛОГИЧЕСКИМ. Удаляемые из операнда разряды последовательно помещаются в бит **CF** регистра флагов. **Эти команды позволяют ОЧЕНЬ быстро выполнять деление (при сдвиге ВПРАВО — R) и умножение (при сдвиге ВЛЕВО — L) операнда на степени двойки.**

7.2.1. Команды арифметического сдвига SAL и SAR

Команды **SAL** (Shift Arithmetic operand Left — сдвиг арифметического операнда влево) и **SAR** (Shift Arithmetic operand Right — сдвиг арифметического операнда вправо) в основном предполагалось применять для **знаковых** целочисленных данных. Но операция **SAL** совершенно идентична команде **SHL**. Команда **SAR** хотя и репродуцирует знак, но, если сдвигается отрицательное нечетное число, результат получается не совсем точный — см. пример 7.2 (**test #1, test #3**).

Таблица 7.7. Операции побитового линейного сдвига в алгоритмических языках Паскаль и С/С++.

Операции линейного сдвига	Обозначение в алгоритмических языках		Примеры применения в алгоритмических языках	
	С/С++	Паскаль	С/С++	Паскаль
Сдвиг влево	<<	SHL	A << 5	A SHL 5
Сдвиг вправо	>>	SHR	B >> 3	B SHR 3

Флаг CF	Знаковый бит (Signum)	Остальные информационные биты	Младший бит (0)	
←			←	0

РИС. 7.1. *Логика работы команды арифметического сдвига влево SAL.*

Знаковый бит	Информационные биты				Флаг CF
S	Бит S-1	...	1	0	
	↑	→			→

0 (1 — для отрицательных чисел)

РИС. 7.2. *Логика работы команды арифметического сдвига вправо SAR.*

Операции сдвига — это побитовые операции и они прекрасно работают именно в этом качестве! Если же вы хотите их использовать для ускорения арифметических операций деления и умножения, нужно хорошо представлять себе допустимый диапазон как вводимых данных, так и результата (см. примеры 7.2 и 7.3).

Поясним выполнение этих команд на конкретном примере.

Флаг CF	Биты в байте							
	7	6	5	4	3	2	1	0
	Исходное число в регистре AL:							
	0	0	1	1	1	0	1	0
	SAL AL,1 ; сдвиг влево на один разряд							
	Получаем результат (в регистре AL):							
0	0	1	1	1	0	1	0	0
	SAL AL,3 ; сдвиг влево еще на три разряда							
	Получаем результат (в регистре AL):							
1	1	0	1	0	0	0	0	0

РИС. 7.3. *Логика работы команды SAL для положительного числа 58.*

Проанализируем результат операции арифметического сдвига ВЛЕВО. В регистре AL хранилось число 58 (00111010). Первый сдвиг дал результат 116 (01110100), т.е. 58*2 = 116. Следующий сдвиг на три разряда влево для данного числа уже НЕКОРРЕКТЕН, поскольку предполагает, что нужно сделать такую операцию: $116*2^3 = 116*8 = 928$. Т.е. результат выходит за диапазон 8-разрядного числа. Если бы исходное число было сразу размещено в регистре AX, все было бы нормально.



ОБРАТИТЕ ВНИМАНИЕ

На **знаковый бит** у команды **SAL** нет НИКАКОЙ реакции: он просто последовательно выталкивается в бит CF регистра флагов.

? ВОПРОС

Что получится, если в регистрах AL/AX будет отрицательное число?

Сделаем аналогичные предыдущему операции, но — вправо. У команды **SAR** в плане реакции на **знаковый бит** дело обстоит более благополучно — знак репродуцируется, т.е. восстанавливается, а нуль поступает последовательно в соседний со знаковым бит (в данном случае в 6-ой).

Биты в байте								**Флаг CF**
7	6	5	4	3	2	1	0	
Исходное число в регистре AL:								
0	0	1	1	1	0	1	0	
SAR AL,1 ; сдвиг вправо на один разряд								
Получаем результат (в регистре AL):								
0	0	0	1	1	1	0	1	0
SAR AL,3 ; сдвиг вправо еще на три разряда								
Получаем результат (в регистре AL):								
0	0	0	0	0	0	1	1	1

РИС. 7.4. *Логика работы команды SAR для положительного числа 58.*

Проанализируем результат операции арифметического сдвига ВПРАВО. Итак, в регистре AL хранилось число 58 (00111010). Первый сдвиг дал результат 29 (00011101), т.е. 58/2 = 29. Следующий сдвиг на три разряда вправо для данного числа предполагает, что нужно сделать такую операцию: $29/2^3 = 29/8 = 3$. Что мы и получили.

? ВОПРОС

Что получится, если и в этом случае в регистрах AL/AX будет отрицательное число?

ПРИМЕР 7.2.

Выполнить следующие вычисления: **y:=16*x; z:=w/2;** для знаковых 16-разрядных данных. Сделаем реализацию этой задачи в связке Паскаль+Ассемблер. Заметим, что $16=2^4$. Произведем анализ полученных результатов.

Исходный текст программы shift1.pas

```
program shiftpascal;
{CopyRight by Голубь Н.Г. 1993-1998
 y:=16*x; z:=w/2; для знаковых 16-разрядных данных
 !!!!!!!КОНТРОЛЬ ДИАПАЗОНА ОТСУТСТВУЕТ!!!!!!!
 }
{$I shift1a}
{$f+}
USES CRT;
var x,w,y,z,t :Integer;
      ch       :char;
Procedure shift1; external;
Procedure shiftp1;
Begin
  y:=16*x;    {Можно и так:    y:=x shl 4;}
{z:=w shr 1; — РЕЗУЛЬТАТ НЕ ВСЕГДА КОРРЕКТНЫЙ!!! Особенно для чисел < 0}
  z:=w div 2;
End;
Begin
  ch:='y';
  while (ch='y') or (ch='Y') do
   begin
     inc(t);
    Writeln ('=================== test #', t,' ============================');
     Writeln('Вычислить: y:=16*x; z:=w/2; для знаковых 16-разрядных данных');
     write ('введите x,w: [-32768..32767] ====> ');
     readln (x,w);
    shift1;
    writeln ('АССЕМБЛЕР: y=16*',x,'=',y,';  z=',w,'/2=',z);
    y:=0; z:=0;
    shiftp1;
    writeln ('ПАСКАЛЬ:   y=16*',x,'=',y,';  z=',w,'/2=',z);
    writeln ('продолжать? (y/n)');
    ch:=ReadKey;
   end
End.
```

Исходный текст модуля shift1a.asm

```
          title shift1a
; CopyRight by Голубь Н.Г. 1993-1998
; y:=16*x; z:=w/2; для знаковых 16-разрядных данных
data    segment   para public
        Extrn     x:word,w:word,y:word,z:word
data    Ends
code    segment   para public
        assume    cs:code,ds:data
        Public    shift1
shift1  proc      far
        mov       ax,x
        mov       bx,w
        mov       cl,4
        SAL       ax,cl              ; 16*x
        mov       y,ax
        SAR       bx,1               ; w/2
```

```
        mov     z,bx
        ret
shiftl  endp
code    ends
        end     shiftl
```

Выполним тестовые примеры:

```
===================== test #1 ===========================
Вычислить: y:=16*x; z:=w/2; для знаковых 16-разрядных данных
введите x,w: [-32768..32767] ====> -2000 -30001
АССЕМБЛЕР: y=16*-2000=-32000;  z=-30001/2=-15001
ПАСКАЛЬ:    y=16*-2000=-32000;  z=-30001/2=-15000
продолжать? (y/n)
===================== test #2 ===========================
Вычислить: y:=16*x; z:=w/2; для знаковых 16-разрядных данных
введите x,w: [-32768..32767] ====> 20000 30001
АССЕМБЛЕР: y=16*20000=-7680;  z=30001/2=15000
ПАСКАЛЬ:    y=16*20000=-7680;  z=30001/2=15000
продолжать? (y/n)
===================== test #3 ===========================
Вычислить: y:=16*x; z:=w/2; для знаковых 16-разрядных данных
введите x,w: [-32768..32767] ====> 2000 -5
АССЕМБЛЕР: y=16*2000=32000;  z=-5/2=-3
ПАСКАЛЬ:    y=16*2000=32000;  z=-5/2=-2
продолжать? (y/n)
===================== test #4 ===========================
Вычислить: y:=16*x; z:=w/2; для знаковых 16-разрядных данных
введите x,w: [-32768..32767] ====> -222 -33333
АССЕМБЛЕР: y=16*-222=-3552;  z=32203/2=16101
ПАСКАЛЬ:    y=16*-222=-3552;  z=32203/2=16101
продолжать? (y/n)
===================== test #5 ===========================
Вычислить: y:=16*x; z:=w/2; для знаковых 16-разрядных данных
введите x,w: [-32768..32767] ====> -6372 11111
АССЕМБЛЕР: y=16*-6372=29120;  z=11111/2=5555
ПАСКАЛЬ:    y=16*-6372=29120;  z=11111/2=5555
продолжать? (y/n)
```

Из этих тестовых примеров четко видно следующее:

- **ПРОГРАММИСТ должен следить за диапазоном как вводимых, так и выводимых данных (test #2, test #4, test #5).**
- **Операция SAR корректно работает с положительными целыми числами и НЕ всегда корректно — с ОТРИЦАТЕЛЬНЫМИ НЕЧЕТНЫМИ числами (test #1, test #3).**

Именно из-за таких подводных камней команды арифметического сдвига применяются достаточно редко.

7.2.2. Команды логического сдвига SHL и SHR

Команда **SHL** абсолютно идентична команде **SAL**.

Флаг CF	Информационные биты				
	Старший	...	1	0	
←	←			←	0

РИС. 7.5. *Логика работы команды логического сдвига влево SHL.*

	Информационные биты				Флаг CF
	Старший	...	1	0	
0	→			→	CF

РИС. 7.6. *Логика работы команды логического сдвига вправо SHR.*

Рассмотрим на примере работу команды **SHR**.

Биты в байте								Флаг CF
7	6	5	4	3	2	1	0	
Исходное число в регистре AL:								
1	0	1	1	1	0	1	0	
SHR AL,1 ; сдвиг вправо на один разряд								
Получаем результат (в регистре AL):								
0	1	0	1	1	1	0	1	0
SHR AL,3 ; сдвиг вправо еще на три разряда								
Получаем результат (в регистре AL):								
0	0	0	0	1	0	1	1	1

РИС. 7.7. *Логика работы команды SHR.*

ПРИМЕР 7.3.

Выполнить следующие вычисления: **y:=16*x; z:=w/2;** для БЕЗзнаковых 16-разрядных данных. Сделаем реализацию этой задачи тоже в связке Паскаль+Ассемблер. Заметим, что $16=2^4$. В Паскале можно тоже делать *логические* сдвиги, воспользовавшись операциями **shl** и **shr**. Произведем анализ полученных результатов.

Исходный текст программы shift2.pas

```
program shiftpascal;
{CopyRight by Голубь Н.Г. 1993-1998
 y:=16*x; z:=w/2; для БЕЗзнаковых 16-разрядных данных
 !!!!!!!КОНТРОЛЬ ДИАПАЗОНА ОТСУТСТВУЕТ!!!!!!!
 }
{$I shift2a}
{$f+}
USES CRT;
```

```
var  x,w,y,z,t        :Word;
     ch               :char;
Procedure shift2; external;

Procedure shiftp2;
Begin
  y:=x shl 4;   {Так быстрее, чем        y:=16*x;}
  z:=w shr 1;   {Так быстрее, чем        z:=w div 2;}
End;

Begin
  ch:='y';
  while (ch='y') or (ch='Y') do
   begin
     inc(t);
    Writeln ('================== test #', t,' ===========================');
     Writeln('Вычислить: y:=16*x; z:=w/2; для БЕЗзнаковых 16-разрядных данных');
     write ('введите x,w: [0..65535] ====> ');
     readln (x,w);
     shift2;
     writeln ('АССЕМБЛЕР: y=16*',x,'=',y,';  z=',w,'/2=',z);
     y:=0; z:=0;
     shiftp2;
     writeln ('ПАСКАЛЬ:   y=16*',x,'=',y,';  z=',w,'/2=',z);
     writeln ('продолжать? (y/n)');
    ch:=ReadKey;
  end
End.
```

Исходный текст модуля shift2a.asm

```
        title shift2a
; CopyRight by Голубь Н.Г. 1993-1998
; y:=16*x; z:=w/2; для БЕЗзнаковых 16-разрядных данных
data    segment   para public
        Extrn     x:word,w:word,y:word,z:word
data    Ends
code    segment   para public
        assume    cs:code,ds:data
        Public    shift2
shift2  proc      far
        mov       ax,x
        mov       bx,w
        mov       cl,4
        SHL       ax,cl       ; 16*x
        mov       y,ax
        SHR       bx,1        ; w/2
        mov       z,bx
        ret
shift2  endp
code    ends
        end       shift2
```

Тестовые примеры:

```
====================== test #1 ===========================
Вычислить: y:=16*x; z:=w/2; для БЕЗзнаковых 16-разрядных данных
введите x,w: [0..65535] ====> 2000 55555

АССЕМБЛЕР: y=16*2000=32000;   z=55555/2=27777
ПАСКАЛЬ:    y=16*2000=32000;   z=55555/2=27777

продолжать? (y/n)
====================== test #2 ===========================
Вычислить: y:=16*x; z:=w/2; для БЕЗзнаковых 16-разрядных данных
введите x,w: [0..65535] ====> 3000 66666

АССЕМБЛЕР: y=16*3000=48000;   z=1130/2=565
ПАСКАЛЬ:    y=16*3000=48000;   z=1130/2=565

продолжать? (y/n)
====================== test #3 ===========================
Вычислить: y:=16*x; z:=w/2; для БЕЗзнаковых 16-разрядных данных
введите x,w: [0..65535] ====> -4444 20001

АССЕМБЛЕР: y=16*61092=59968;   z=20001/2=10000
ПАСКАЛЬ:    y=16*61092=59968;   z=20001/2=10000

продолжать? (y/n)
```

Попробуйте теперь самостоятельно сделать выводы...

7.2.3. Команды циклического сдвига ROL, ROR, RCL и RCR

Эти команды, в отличие от предыдущих, *сохраняют* значения сдвигаемых бит и очень широко используются, например, при проверке участка памяти (или всей памяти целиком) компьютера. Различают команды *простого* циклического сдвига: **ROL** или **ROR** (ROtate Left или ROtate Right) и циклического сдвига через флаг переноса: **RCL** или **RCR** (Rotate through Carry Left или Rotate through Carry Right).

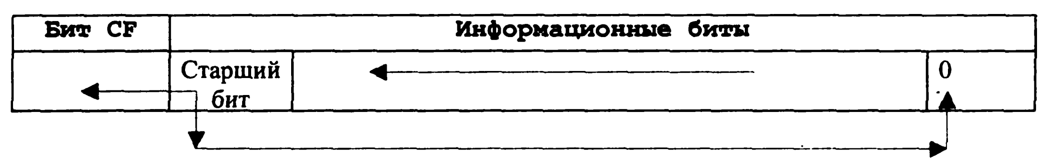


РИС. 7.8. *Логика работы команды простого циклического сдвига влево ROL.*

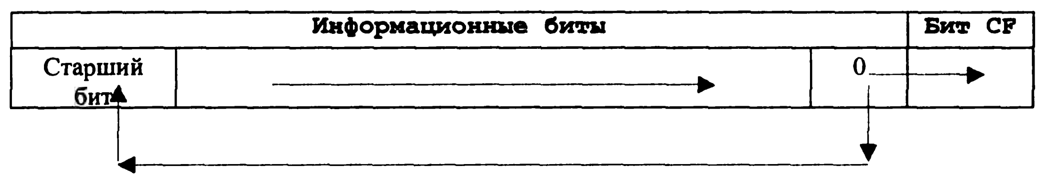


РИС. 7.9. *Логика работы команды простого циклического сдвига вправо ROR.*

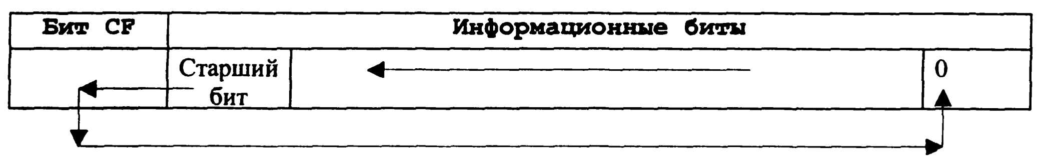


РИС. 7.10. *Логика работы команды циклического сдвига влево RCL — через бит CF.*

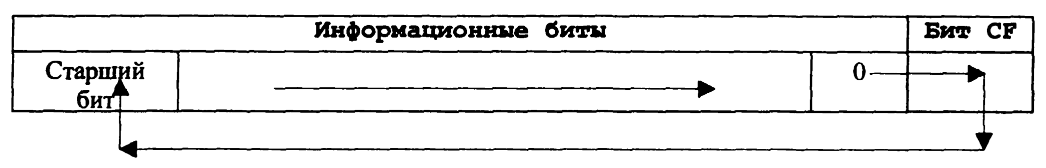


РИС. 7.11. *Логика работы команды циклического сдвига вправо RCR — через бит CF.*

8

Введение в машинные коды IBM PC XT/AT

Лучше сказать лишнее,
чем не сказать необходимого.

Плиний Младший
(61 или 62 — ок. 114 гг.)

Познакомимся теперь с самими машинными кодами и попробуем понять принцип функционирования процессоров Intel семейства IBM PC на примере простейших, уже рассмотренных нами команд Ассемблера. Это научит вас видеть со временем узкие места в своих программах. А потом, кто знает, может, и писать свои компиляторы...

Мы уже знаем, что человек общается с компьютером с помощью языков (алгоритмических или машинно-ориентированных типа языка Ассемблера), а компиляторы переводят операторы (команды) этого языка в машинный код. Как же формируется машинный код, от чего зависит длина программы? На эти и подобные им вопросы мы и попытаемся ответить в настоящей главе и во всех последующих.

8.1. Формат команд процессоров i8086/i8088/i286

С понятием *машинная команда* Ассемблера мы впервые встретились в п. 2.4.1. Там же мы узнали, что ее назначение — максимально приблизить к человеку возможности программирования с учетом внутренней архитектуры процессора Intel. Ключевым моментом в машинной команде Ассемблера был *мнемонический код операции* (*мнемокод, МКОП*).

Машинным кодом мы будем называть внутреннее представление команд Ассемблера в памяти компьютера, т.е. **собственно команды процессора Intel**. Эти команды тоже имеют определенную структуру, состоящую в общем случае из восьми элементов — полей (см. рис. 8.1). Каждое поле имеет длину 1 байт. В машинном коде ключевым полем, конечно, является **байт кода операции** (поле 3), остальные элементы являются необязательными. Максимальное число полей реальной команды обычно не превышает шести.

Поля команды процессора Intel							
1	2	3	4	5	6	7	8
Байты префикса		**Байт кода операции**	1) Байт способа адресации. 2) Вторичный код операции.	1) Смещение (0,1 или 2 байта) – если один из операндов находится в оперативной памяти или стеке. 2) Непосредственный операнд (0,1,2 или 4 байта): константы, номер порта или прерывания.			

РИС. 8.1. *Общий формат команды процессора Intel.*

Коды операций команд Ассемблера для процессоров i8086/i8088/i286 приведены в приложении 7.

8.1.1. Байт кода операции

Что же собой представляет байт кода операции? Как процессор знает, что ему надо делать? Байт кода операции может иметь несколько модификаций. Рассмотрим их и укажем основные операции, которые они поддерживают. Кодировка кода операции для различных команд Ассемблера приведена в табл. П7.1.

Биты	7	6	5	4	3	2	1	0
	КОП						d	w
							s	w
							v	w
								z

РИС. 8.2. *Вариант 1 — большинство двухадресных команд.*

Биты 2-7 определяют информационную часть кода операции (КОП).

Бит 0 может интерпретироваться двояко:

- **длина операндов:**
  w=1 — обрабатывается слово; **w=0** — обрабатывается байт,
- **значение флага ZF при использовании префикса повторения REP:**
  z=1 — флаг **ZF=1** (команды **REPZ/REPE**);
  z=0 — флаг **ZF=0** (команды **REP/REPNE**).

Бит 1 — может иметь три интерпретации

- **задавать бит направления передачи информации** (в этом случае в команде ОБЯЗАТЕЛЬНО присутствует *байт способа адресации* — см. п. 8.1.2, рис. 8.7):
  d=1 — r/m ===> reg;
  d=0 — r/m <=== reg. Значение reg определяется по табл. 8.1.
- **бит размера непосредственного операнда:**
  s=1 — длина данного 8 бит;
  s=0 — длина данного 16 бит,

- **бит, определяющий значение счетчика в циклических командах:**
  v=1 — значение счетчика находится в регистре CL;
  v=0 — значение счетчика равно 1.

Таблица 8.1. Кодировка регистров.

reg	w=0	w=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Биты	7	6	5	4	3	2	1	0
	КОП				w	reg		

РИС. 8.3. *Вариант 2 — команды загрузки непосредственных данных в регистр reg.*

Значение битов **w** и **reg** аналогично предыдущему.

Биты	7	6	5	4	3	2	1	0
	КОП					reg		

РИС. 8.4. *Вариант 3 — команды загрузки в регистр reg при работе со стеками, одноадресные команды.*

Биты	7	6	5	4	3	2	1	0
	0	0	0	rs		1	1	d

РИС. 8.5. *Вариант 4 — команды работы с сегментными регистрами.*

Бит 0 задает бит направления передачи информации (в этом случае в команде ОБЯЗАТЕЛЬНО присутствует *байт способа адресации* — см. п. 8.1.2, рис. 8.8):
d=1 — r/m ===> rs;
d=0 — r/m <=== rs. Значение rs определяется по табл. 8.2..

Таблица 8.2. Кодировка сегментных регистров.

rs	Сегментные регистры
00	ES
01	CS
10	SS
11	DS

Биты	7	6	5	4	3	2	1	0
	КОП							w

РИС. 8.6. *Вариант 5 — команды загрузки аккумулятора AX (AL) в память и наоборот, работа с портами, арифметические операции с непосредственной адресацией и аккумулятором AX (AL), команды деления и умножения.*

8.1.2. Байт способа адресации

Байт способа адресации (БСА) присутствует в машинном коде только в том случае, если в байте кода операции недостаточно информации об адресации операндов (обычно это двухадресные команды) или если команда имеет вторичный код операции (см. приложение 7).

Байт способа адресации имеет три разновидности.

Биты	7	6	5	4	3	2	1	0
	mod		reg			r/m		

РИС. 8. 7. *Байт способа адресации для обычных двухадресных команд.*

Биты	7	6	5	4	3	2	1	0
	mod		0	rs		r/m		

РИС. 8. 8. *Байт способа адресации для команд, работающих с сегментными регистрами.*

Биты	7	6	5	4	3	2	1	0
	mod		ВторКОП			r/m		

РИС. 8. 9. *Байт способа адресации для команд со вторичным кодом операции.*

Вторичные коды операции приведены в табл. П7.2.

Биты 6-7 (mod) задают режим адресации. В зависимости от их значения процессор распознает значения **битов 0-2 (r/m)** — см. табл. 8.3 или табл. 8.1. В зависимости от бита направления **d** (см. рис. 8.2 или рис. 8.5) определяется, какой из операндов является приемником, а какой источником.

Если **mod=11**, **r/m=reg**, т.е. и приемник и источник являются регистрами (см. табл. 8.1). В этом случае имеем дело с *регистровой адресацией*. Как правило, это — самый быстрый способ адресации, который дает и короткий код. Но регистров не так уж много, поэтому нужен разумный компромисс.

Смещение имеет длину 1 байт, если **mod=01**, и 2 байта, если **mod=10**.

Таблица 8.3. Режимы адресации.

mod	r/m	Формат операнда	Название режима адресации
00	000	[BX+SI]	Регистровый косвенный по базе BX или BP с индексированием по SI или DI (базово-индексный)
	001	[BX+DI]	
	010	[BP+SI]	
	011	[BP+DI]	
	100	[SI]	Регистровый косвенный по индексу SI или DI (индексный)
	101	[DI]	
	110	смещение	Прямой (относительный)
	111	[BX]	Регистровый косвенный по базе BX (базовый)
01 10	000	[BX+SI] + смещение	По базе BX или BP с индексированием по SI или DI и смещением (базово-индексный со смещением)
	001	[BX+DI] + смещение	
	010	[BP+SI] + смещение	
	011	[BP+DI] + смещение	
	100	[SI] + смещение	Прямой (относительный) с индексированием по SI или DI
	101	[DI] + смещение	
	110	[BP] + смещение	По базе BX или BP со смещением
	111	[BX] + смещение	

8.2. Простейшие примеры ассемблирования

Поясним изложенное выше на примерах. Для этого нам понадобится файл листинга — **LST-файл**. Мы с ним уже работали, когда изучали внутреннее представление данных (см. пп. 2.4.4 и 2.4.5). Теперь пришло время познакомиться с внутренним представлением команд Ассемблера, т.е. с самими машинными кодами.

Обратимся к уже известному нам **примеру 6.2** (см. пп. 6.1.2.3, 6.1.2.4) — вариант 2.

Файл листинга PRIM.LST

```
Turbo Assembler    Version 4.1        19/08/01 14:10:44
      Page 1
PRIM.ASM
Арифметические выражения
  1                   title Арифметические выражения
  2    0000           model  large,C
  3                   ; tasm PRIM.ASM /l  /ml!!!!!!!!!!!!!!!!
  4                   ; CopyRight by Голубь Н.Г., 1993-1997, 2000
```

```
                                    ; Пример 6.2.
                                    ;x=(2*a + b*c)/(d-a), d<>a !!!
                                    ;int x,a,b,c,d;
     0000                           DataSeg
                                  ; ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
                                    Extrn    C X:word
                                    Extrn    C a:word
     0000                           CODESEG
                                  ; ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ - КОНСТАНТЫ
     0000 FEB3                      b     dw           -333
     0002 03E8                      c     dw           1000
     0004 FFF6                      d     dw           -10
                                          Public    C prim
     0006                           prim  proc      far
     0006 B8  0002                  mov   ax,2
     0009 F7  2E  0000e             Imul  a       ; <dx>:<ax>=2*a
     000D 8B  DA                    mov   bx,dx   ; bx <=== ст.часть   (dx)
     000F 8B  C8                    mov   cx,ax   ; cx <=== мл.часть   (ax)
     0011 2E: A1  0000r             mov   ax,b
     0015 2E: F7  2E 0002r          Imul  c       ; <dx>:<ax>=b*c
     001A 03  C1                    add   ax,cx   ; <ax>=<ax>+<cx>    (мл.часть)
     001C 13  D3                    adc   dx,bx   ; <dx>=<dx>+<bx>    (ст.часть)
     001E 2E: 8B  0E  0004r         mov   cx,d
     0023 2B  0E  0000e             sub   cx,a    ; <cx>=<cx>-a
     0027 F7  F9                    Idiv  cx      ; <ax>=<dx>:<ax>/<cx>
     0029  A3  0000e                mov   X,ax
1    002C  CB                       RET   00000h
     002D                           prim  endp
                                    end
Turbo Assembler Version 4.1 19/08/01 14:10:44      Page 2
  Symbol Table
  Арифметические выражения
  Symbol Name                  Type      Value
  ??date                       Text      "19/08/01"
  ??filename          Text   "PRIM        "
  ??time                       Text      "14:10:44"
  ??version           Number 040A
  @32Bit                       Text      0
  @CodeSize                    Text      1
  @Cpu                         Text      0101H
  @DataSize           Text   1
  @FileName           Text   PRIM
  @Interface          Text   001h
  @Model                       Text      5
  @WordSize                    Text      2
  @code                        Text      PRIM_TEXT
  @curseg                      Text      PRIM_TEXT
  @data                        Text      DGROUP
  @stack                       Text      DGROUP
  X  (_X)                      Word      DGROUP:— Extern
  a  (_a)                      Word      DGROUP:— Extern
  b                     Word PRIM_TEXT:0000
  c                     Word PRIM_TEXT:0002
  d                     Word PRIM_TEXT:0004
```

```
prim (_prim)                  Far     PRIM_TEXT:0006
Groups & Segments             Bit Size Align  Combine Class
DGROUP                        Group
  _DATA                       16  0000 Word       Public  DATA
PRIM_TEXT                16   002D Word       Public  CODE
```

Сделаем предварительный анализ этого файла.

1. Внешние (**External**) данные **X** и **a**, передаваемые из программы на C++, хранятся в сегменте данных:

X (_X)	Word	DGROUP:— Extern
a (_a)	Word	DGROUP:— Extern

и их адрес (до момента компоновки и выполнения) НЕ известен. Поэтому всюду, где в командах встречаются эти переменные, в кодах стоит смещение **0000e** (см. строки 20, 28 и 30 в файле листинга).

2. Есть также и внутренние (локальные) переменные **b**, **c**, **d**, которые хранятся в сегменте кода. Их адреса выражены относительными (**Relative**) смещениями:

b	Word	PRIM_TEXT:0000
c	Word	PRIM_TEXT:0002
d	Word	PRIM_TEXT:0004

Всюду, где в командах встречаются эти переменные, в кодах стоит соответствующее значение смещения с буквой **r** (см. строки 23, 24 и 27 в файле листинга), а перед командой стоит **префикс 2E**. Из табл.П7.1 мы видим, что кодом **2E** обозначена команда **SEG CS**. Дело в том, что по умолчанию данные размещаются в сегменте данных. Если это не так (как в нашем случае), при компиляции появляется соответствующий префикс.

3. Длина нашего модуля 002D (45) байт.

4. В этом примере есть команды с регистровой, прямой и непосредственной адресацией. Разберемся с некоторыми из них — в файле листинга они выделены.

   - **Строка 19 0006 B8 0002 mov ax,2**

Адрес (смещение) данной команды равен 0006. Пока адрес нам НЕ нужен.

КОП = B8. Он соответствует команде **MOV AX,im16** (см. Табл. П7.1). Команда имеет *непосредственный режим адресации* (16-разрядная константа непосредственно встраивается в машинную команду). Распишем КОП в двоичной системе счисления: 1011 1000 и попробуем понять, какой из вариантов байта кода операции нам подходит (см. п. 8.1.1). Первый вариант нам НЕ подходит, поскольку **бит 0 (w)=0** (наша команда НЕ работает с 8-разрядными данными). Вариант 2 нам подходит: **w=1**, **reg=000=AX** (см. табл. 8.1):

Биты	7	6	5	4	3	2	1	0
	1	0	1	1	1	0	0	0
	КОП				w	reg		

? ВОПРОС

Что изменилось бы в машинном коде, если бы наша операция выглядела так:
mov ax,-22

- **Строка 21 000D 8B DA mov bx,dx**

Здесь мы имеем дело с *регистровой адресацией*. **КОП = 8В**. Он соответствует команде **MOV r16,r/m** (см. Табл. П7.1). Это обычная двухадресная команда, у которой есть байт способа адресации (БСА). Распишем КОП и БСА в двоичной системе счисления: 1000 1011 1101 1010 и попробуем их расшифровать. Это — однозначно вариант 1 (см. рис. 8.2 и 8.7):

	Байт кода операции										**Байт способа адресации**							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	1	0	0	0	1	0	1	1			1	1	0	1	1	0	1	0
	КОП						d	w			mod		reg			r/m		

Проанализируем БСА совместно с КОП. Из табл. 8.3 получаем: **mod=11**, значит, **r/m=reg=010=DX** (**w=1**, см. табл. 8.1). Значение собственно поля **reg=011=BX**. Бит **d=1**, значит, регистр **BX** является приемником. Вот мы с вами и сделали *ассемблирование*. Это же делает и компилятор! Как видите, НИЧЕГО страшного в этом ассемблировании нет! Мы с вами немного поработали компиляторами...

? ВОПРОС

Что изменилось бы в машинном коде, если бы наша операция выглядела бы так:
mov dx, bx ?

- **Строка 23 0011 2E: A1 0000r mov ax,b**

Здесь мы имеем дело с *прямой (относительной) адресацией* (один из операндов — область памяти, смещение записывается прямо в машинный код). **КОП = A1**. Он соответствует команде **MOV AX, mem16** (см. Табл. П7.1). Здесь мы имеем дело с командой загрузки аккумулятора AX из памяти. Распишем КОП в двоичной системе счисления: 1010 0001 и попробуем его расшифровать. Это — однозначно вариант 5 (см. рис. 8.6):

Биты	7	6	5	4	3	2	1	0
	1	0	1	0	0	0	0	1
	КОП							w

? ВОПРОС

Что изменилось бы в машинном коде, если бы наша операция выглядела так:
mov aL, BYTE PTR b

- **Строка 25 001A 03 C1 add ax,cx**

Это тоже *регистровая адресация*, команда содержит байт кода операции и байт способа адресации. Идейно реализация данной команды НИЧЕМ НЕ отличается от команды в строке 21, которую мы уже рассматривали. Проделаем те же выкладки и в результате получим:

	Байт кода операции										Байт способа адресации							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	1	1			1	1	0	0	0	0	0	1
	КОП						d	w			mod		reg			r/m		

Проанализируем БСА совместно с КОП. Из табл. 8.3 следует: **mod=11**, значит, **r/m=reg=001=CX** (**w=1**, см. табл. 8.1). Значение собственно поля **reg=000=AX**. Бит **d=1**, значит, регистр **AX** является приемником.

? ВОПРОС

Что изменилось бы в машинном коде, если бы наша операция выглядела так:

add aL,cL

- **Строка 28 0023 2B 0E 0000e sub cx,a**

Эта команда очень похожа на предыдущую. Здесь тоже есть наряду с КОП и БСА, но здесь другой режим адресации — *прямая адресация*, т.е. в коде появляется еще и смещение (как в строке 23), но только смещение с буквой **e** (переменная внешняя и описана, как полагается по умолчанию, в сегменте данных). Проделаем уже известные рассуждения и получим следующее:

	Байт кода операции										Байт способа адресации							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	0	0	1	0	1	0	1	1			0	0	0	0	1	1	1	0
	КОП						d	w			mod		reg			r/m		

Проанализируем БСА совместно с КОП. Из табл. 8.3 получим: **mod=00**, значит, **r/m=110=смещение**. Значение поля **reg=001=CX** (**w=1**, см. табл. 8.1). Бит **d=1**, значит, регистр **CX** является приемником.

? ВОПРОС

Что изменилось бы в машинном коде, если бы наша операция выглядела так:

sub Byte PTR a, cL

- **Строка 29 0027 F7 F9 Idiv cx**

Здесь тоже все понятно с машинным кодом, не правда ли? Не торопитесь... Хотя команда и одноадресная, но она имеет вторичный код операции **Grp1 r16/m16** (см. табл. П7.1). Значит, нам надо еще поискать его в табл. П7.2 и зависит он от содержимого битов БСА.

Байт кода операции										Байт способа адресации							
7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1			1	1	1	1	1	0	0	1
КОП						d	w			Mod		ВторКОП			r/m		

Проанализируем БСА совместно с КОП. Из табл. 8.3 получим: **mod=00**, значит, **r/m=reg=001=CX** (**w=1**, см. табл. 8.1). Значение поля **ВторКОП=111**. Значит, перед нами команда **IDIV** (см. табл. П7.2). Бит **d=1**, но о приемнике в этой команде речи нет (она одноадресная и, как мы знаем, достаточно хитрая. Не забыли еще, где хранится результат?). Поэтому КОП в данном случае надо интерпретировать не по варианту 1, а по варианту 5. Окончательная расшифровка данной команды будет иметь вид:

	Байт кода операции										Байт способа адресации							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	1	1	1	1	0	1	1	1			1	1	1	1	1	0	0	1
	КОП							w			mod		ВторКОП			r/m		

Да, недаром команда **IDIV** "хитрая"!

? ВОПРОС

Что изменилось бы в машинном коде, если бы наша операция выглядела так:

div bx

- **Строка 24 0015 2E: F7 2E 0002r Imul c**

Ну, теперь-то уж, надеюсь, все понятно! Действительно, эта команда ОЧЕНЬ похожа на предыдущую во всех отношениях (даже основной код операции у них одинаковый!), кроме режима адресации: у предыдущей был регистровый режим адресации, а у нашей — *прямой*. И в этом плане настоящая команда еще похожа на команду в строке 23 (у нее тоже есть префикс! Что это значит, помните?).

	Байт кода операции										Байт способа адресации							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	1	1	1	1	0	1	1	1			0	0	1	0	1	1	1	0
	КОП							w			mod		ВторКОП			r/m		

? ВОПРОС

Что изменилось бы в машинном коде, если бы наша операция выглядела так:

MUL a

9

Команды передачи управления для IBM PC XT/AT

Глупый верит всякому слову, благоразумный же внимателен к путям своим.

Притчи Соломона (гл. 14, ст.15)

До сих пор мы изучали так называемые линейные команды, которые выполняются последовательно друг за другом. Какая команда выполняется следующей, об этом всегда знает регистр IP (см. п. 3.3.3). **Команды передачи управления позволяют нарушить естественную последовательность выполнения команд путем изменения содержимого регистра IP**. В данной главе мы и рассмотрим **базовые команды** передачи управления. Это — команды безусловного перехода на определенный адрес в памяти компьютера, команды условного перехода в зависимости от результата проверки условия и команды организации циклических вычислений. **Команды передачи управления НЕ меняют значения флагов**.

9.1. Команды безусловной передачи управления

Мы познакомимся с тремя основными командами безусловной передачи управления. Самая простая из них — команда JMP. Команды CALL и RET чуть более сложные и являются взаимно обратными командами.

9.1.1. Команда безусловного перехода JMP

Это, пожалуй, наиболее распространенная команда в Ассемблере, которая является аналогом самого многострадального (по количеству нападок на него) и известного в алгоритмических языках оператора:

```
goto метка;
```

Вообще в таких языках, как C++ или Паскаль, в большинстве случаев прекрасно можно обойтись без этого оператора. Он считается потенциальной угрозой и поклонники *структурного программирования* его не применяют вообще. Например, в языке Java такое ключевое слово есть, но оно НЕ поддерживается. Вот что пишет **Б. Страуструп** по этому поводу: *Существует очень мало примеров использования пресловутой инструкции* **goto** *при программировании на высоком уровне, но эта инструкция может быть крайне полезна, когда код C++ генерируется программой, а не человеком.* Однако начинающие программисты очень любят этот оператор. Понимание, как можно обойтись без этой *пресловутой инструкции*, приходит со временем и с опытом знакомства с другими инструкциями C++ или Паскаль... Мы, если вы заметили, мои наблюдательные читатели, в своих примерах эту инструкцию НЕ игнорировали и, как вы теперь понимаете, делали это намеренно.

Мнемокод команды безусловной передачи управления **JMP** получен в результате сокращения слова: JuMP — Прыжок. Команда эта содержит **один операнд** и имеет следующий формат (**синтаксис**):

JMP *Метка*

Метка, как и в алгоритмических языках, помечает нужную команду, на которую следует передать управление, и может иметь атрибут **NEAR** (по умолчанию) или **FAR**. Если *Метка* имеет атрибут NEAR, безусловный переход осуществляется в пределах данного сегмента. В этом случае **логика работы команды** следующая:

<IP>= <IP> + *смещение_к_нужной_команде*

Таким образом, автоматически происходит выполнение команды, помеченной *меткой*.

Поскольку в данном случае переход осуществляется в пределах сегмента, *смещение_к_нужной_команде* НЕ может превышать двух байт. Нужная нам команда может располагаться в сторону больших адресов, тогда говорят, что переход осуществляется **вперед**. В этом случае *смещение_к_нужной_команде* — **положительное**. Если, наоборот, нужная нам команда располагается в сторону меньших адресов, говорят о переходе **назад**. В этом случае *смещение_к_нужной_команде* — **отрицательное**.

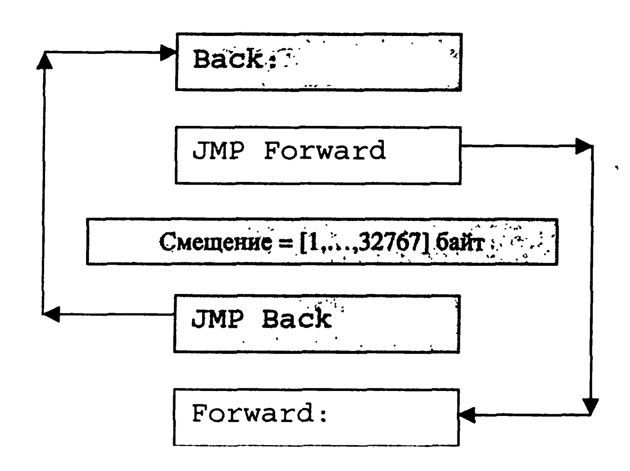


РИС. 9.1. *Схема работы команд JMP Forward (переход вперед) и Jmp Back (переход назад) в пределах одного сегмента (NEAR переход).*

Можно сэкономить один байт при реализации команды безусловной передачи управления, если сделать переход коротким (**SHORT**), т.е. смещение в пределах [-128...127]. В этом случае переход вперед нужно записать таким образом:

JMP **SHORT** Forward

Для передачи управления назад слово **SHORT** можно НЕ писать, потому что компилятор при построении машинного кода сам вычислит, какое получается смещение, поскольку на момент компиляции команды перехода **JMP Back** адрес, на который указывает метка **Back**, ему уже известен (просмотр исходного модуля делается сверху вниз). По умолчанию компилятор делает один проход. Если задать более одного прохода компиляции с помощью ключа **/m#** (например, два прохода **/m2**), компилятор сам сделает там, где нужно, короткий переход — см. пример 9.1.

Теперь разберемся с более сложным случаем, когда надо передать управление метке **Lbl**, находящейся в другом сегменте (**FAR переход**). В этом случае, с точки зрения компьютера, меняется не только содержимое регистра IP, но и содержимое сегментного регистра CS. Воспользовавшись уже знакомыми нам директивами Extern и Public, записать такой переход на Ассемблере можно следующим образом,:

```
........................
Code1  Segment
Extrn  Lbl:FAR
........................
JMP  Code2:Lbl
........................
Code1  EndS
Code2  Segment
Public  Lbl
........................
Lbl:
........................
Code2  EndS
```

В этом случае **логика работы команды JMP** следующая:

<**CS**>=<Code2>

<**IP**>= <**IP**> + *смещение_к_нужной_команде*

Таким образом, автоматически происходит выполнение команды, помеченной *меткой* **Lbl**, находящейся в другом кодовом сегменте (FAR-переход). Все рассуждения относительно смещения аналогичны Near-переходу, только смещение здесь всегда 16-разрядное (для реального режима работы процессора).

ПРИМЕР 9.1.

Поясним сказанное на примере организации переходов и проанализируем **LST**-файлы.

Файл листинга PRIM1.lst

```
Turbo Assembler     Version 4.1     20/08/01 22:40:19     Page 1
PRIM1.asm
Безусловный переход (NEAR, SHORT, FAR)
 1                        ; tasm PRIM2.ASM /l
 2                        ; CopyRight by Голубь Н.Г., 2001
 3                        ; Пример 9.1.
```

```
 4                        ;
 5 0000              Code2 Segment "code";=====================
 6                        Assume cs:Code2
 7 0000              Code2 Ends
 8 0000              Code1 Segment "code";=====================
 9                        Assume cs:Code1
 10                            Extrn Lbl:FAR
 11     0000              jump proc
 12     0000 EB 0A 90          Back: jmp Forward
 13                        ;......................
 14     0003 EB 05             jmp SHORT L2
 15                        ;......................
 16     0005 EA 00000000se     jmp Code2:Lbl
 17                        ;......................
 18     000A EB F4         L2: jmp Back
 19                        ;......................
 20     000C C3            Forward:   ret
 21     000D               jump endp
 22     000D               Code1 EndS
 23                            end
Turbo Assembler    Version 4.1    20/08/01 22:40:19    Page 2
Symbol Table
Безусловный переход (NEAR, SHORT, FAR)
Symbol Name                Type Value
??DATE                     Text "20/08/01"
??FILENAME                 Text "PRIM1    "
??TIME                     Text "22:40:19"
??VERSION                  Number 040A
@CPU                       Text 0101H
@CURSEG                    Text CODE1
@FILENAME                  Text PRIM1
@WORDSIZE                  Text 2
BACK                       Near CODE1:0000
FORWARD                    Near CODE1:000C
JUMP                       Near CODE1:0000
L2                         Near CODE1:000A
LBL                        Far  CODE1:—— Extern
Groups & Segments          Bit Size Align Combine Class
CODE1                      16 000D Para   none  CODE
CODE2                      16 0000 Para   none  CODE
```

Проанализируем строки нашего листинга.

1) Сделан переход вперед на метку **Forward**, сгенерирован машинный код:

12 0000 EB 0A 90 Back: jmp Forward

Вспомним, как мы анализировали код в главе 8. **КОП = EB =JMP short** (см. табл. П7.1). Далее во всех командах перехода следует смещение к нужной нам команде:

20 000C C3 Forward: ret

Мы знаем, что команды перехода изменяют содержимое регистра IP:

<IP>= <IP> + *смещение_к_нужной_команде*

И здесь для анализа нам уже будут необходимы адреса (смещения) команд.

Итак, нам нужно, чтобы содержимое регистра IP изменилось и стало **<IP>=000C**. А какое оно было? Регистр IP всегда указывает на адрес СЛЕДУЮЩЕЙ команды, т.е. **<IP>=0003**. Проверим, так ли это.

2) Теперь посмотрим на смещение к нужной команде: **0A 90**. Что-то уж оно слишком большое. А это потому, что **0A — это действительно смещение** (переход-то короткий, см. КОП!), а *90 — код следующей команды*. Проанализируем его: **КОП=90=NOP** — НЕТ операции (см. табл. П7.1), т.е. это просто ЗАГЛУШКА. Если бы мы сделали явно данный короткий переход **jmp SHORT Forward** или поставили бы ключ компиляции **/m2** (см. файл Prim1m2.lst в материалах, прилагаемых к книге), этой заглушки бы НЕ было, и мы сэкономили бы один байт. Итак, адрес СЛЕДУЮЩЕЙ команды NOP 0002, т.е. <IP>=**0002**.

3) Таким образом, получаем нужный нам адрес (не забывайте, что мы работаем с HEX-кодами):

```
   0002
+    0A
-------
   000C
```

<IP>	=	**<IP>**	+	*смещение_к_нужной_команде*
000C	=	0002	+	0A

<IP>=000C, что и требовалось доказать.

4) Теперь проанализируем переход назад:

18000A EB F4 L2: jmp Back

Теперь нам будет немного легче. Код сгенерирован без всяких заглушек. Адрес следующей команды, хранящийся в регистре IP на момент выполнения команды перехода назад **<IP>=000C**, а смещение отрицательное **F4**:

```
   000C
+    F4
-------
   0000
```

<IP>	=	**<IP>**	+	*Смещение_к_нужной_команде*
0000	=	000C	+	F4

5) Осталось проанализировать команду:

160005 EA 00000000se jmp Code2:Lbl

КОП = EA =JMP far (см. табл. П7.1). Действительно, имеем дальний переход, смещение 32-разрядное **00000000se**, содержащее пока неизвестный на момент компиляции сегмент (буква **s**) и внешнее смещение (буква **e**).

Надеюсь, что с анализом команды

140003 EB 05 jmp SHORT L2

вы и сами управитесь, не правда ли?

9.1.2. Работа с процедурами в Ассемблере

9.1.2.1. Команда вызова процедур CALL

Идейно команда CALL (вызов) аналогична команде JMP. Но поскольку команда CALL предназначена для вызова процедур, дополнительно команда запоминает в стеке еще и адрес точки возврата.

Синтаксис: CALL *Имя_процедуры*

Логика работы команды в случае Near-вызова процедуры:
PUSH IP
<IP>= <IP> + *смещение_к_нужной_процедуре*

Логика работы команды в случае Far-вызова процедуры:
PUSH IP
PUSH CS
<CS>=<Code2>
<IP>= <IP> + *смещение_к_нужной_процедуре*

Если у вызываемой процедуры есть параметры, они передаются через стек ДО вызова процедуры. **Паскаль и C++ по-разному работают с параметрами** (см. п. 9.1.2.3 и п. 9.1.2.4). Для облегчения передачи параметров можно использовать команду **CALL** в **расширенном синтаксисе:**

CALL *Имя_процедуры* [*язык*[, *арг1*,]...],

где *язык* — это **C, CPP, PASCAL, BASIC, FORTRAN, PROLOG, NOLANGUAGE**, *арг* — аргумент, помещаемый в стек в соответствии с принятыми соглашениями.
Пример, иллюстрирующий применение этой директивы, см. в п. 9.6.

9.1.2.2. Команда возврата в точку вызова RET

Команда **RET** (RETurn from procedure — возврат из процедуры) по действию является обратной команде CALL. Она обеспечивает возврат управления вызывающей программе и, если нужно, очистку стека на величину 16-разрядной константы *Im16* байт.

Синтаксис: **RET** [*Im16*]

Логика работы команды в случае Near-вызова процедуры:
POP IP

Логика работы команды в случае Far-вызова процедуры:
POP CS
POP IP

9.1.2.3. Соглашения о вызовах в Borland Pascal

Параметры в стек передаются по порядку слева направо, т.е. первым в стек помещается первый параметр, последним — последний. По окончании работы **вызываемая процедура** должна очистить стек.

Если процедура является функцией (в том понимании, как это принято в языке Паскаль), она должна возвратить значение. Возвращаемые значения должны находиться в соответствующих регистрах в зависимости от их типа:

- **Порядковый тип (integer, word, byte, shortint, char, longInt, boolean**, перечисления):
  - Байт → **AL**
  - Слово → **AX**
  - Двойное слово → **<DX:AX>**
- **Тип Real** → **<DX:BX:AX> (DX** — старшая часть числа, **AX** — младшая часть числа).
- **Single, double, extended** → **ST(0)** — вершина стека математического сопроцессора (см. Главу 13).
- **Pointer** → **<DX:AX> (DX** — сегмент, **AX** — смещение).
- **String** — указатель на временную область памяти.



ЗАМЕЧАНИЕ.

Этот же порядок хранения возвращаемых значений действует и в языках C/C++.

Разберемся с параметрами на конкретном примере.



ПРИМЕР 9.2.

Написать функцию с параметрами для вычисления арифметического выражения **i+j-k** для 16-разрядных целых знаковых или беззнаковых чисел (integer или word).

Вариант 1. Воспользуемся сначала примером, входящим в стандартную поставку TASM (я в нем ничего менять пока НЕ буду).

Исходный текст модуля ASMPARM.ASM — вариант 1.

```
; Turbo Assembler
; Copyright (c) 1988, 1991 By Borland International,
; Inc.
; ASMPARM.ASM
; ——— i+j-k : integer or word ————
; Called as: TESTA(i, j, k);
;
;leftmost parameter — Крайний левый параметр
i       equ  8          ; смещение относительно
j       equ  6          ;         вершины стека
;rightmost parameter — Самый правый параметр
k equ 4
; Вариант 1 — параметры готовит программист
        .MODEL small,pascal
        .CODE
        PUBLIC TESTA
TESTA   PROC
        push bp
        mov  bp,sp
        mov  ax,[bp+i] ;get i
        add  ax,[bp+j] ;add j to i
        sub  ax,[bp+k] ;subtract k from the sum
        pop  bp
        ret  6          ;return, discarding 6 parameter bytes
TESTA   ENDP
        END
```

Программа на Паскале для проверки работы этого модуля для знаковых целочисленных данных находится в прилагаемых к книге материалах.

Попробуем в этом ASM-модуле разобраться. Для этого нам нужно понять процесс передачи фактических параметров, которые должны храниться в стеке. И вообще нужно знать, что находится в стеке на момент выполнения нашей функции.

1. Модель **Small** (см. табл. 3.3) означает, что все процедуры в данном модуле имеют атрибут **NEAR**.
2. Процедуру **TESTA** вызывает Паскаль: **ta:= TESTA(i,j,k);**
3. Мы уже знаем, что ДО вызова процедуры в стек должны быть переданы параметры — в данном случае в естественном для европейца порядке (слева направо: **i**, **j**, **k**):

```
PUSH i
PUSH j
PUSH k
```

Таким образом, о передаче параметров в данном случае заботится тоже Паскаль.

4. Затем следует вызов функции: **CALL TESTA**. Это влечет за собой опять загрузку стека, на этот раз адресом возврата в точку вызова:

```
PUSH IP
```

5. Если вы внимательно просматривали ASM-листинг примера 6.1, полученного с помощью компилятора Borland C++, возможно, обратили внимание на две команды, которые есть и в нашем примере:

```
push bp
mov bp,sp
```

Дело в том, что напрямую работать с вершиной стека НЕ рекомендуется, а работать надо. Поэтому и был предложен вариант использовать для этого регистр **BP**, который и является вершиной стека на момент начала выполнения процедуры. Это соглашение справедливо как для **Borland Pascal**, так и для **Borland C++**. Именно по этой причине я его и не рекомендовала использовать для других целей (см. п. 3.3.7).

Таблица 9.1. Содержимое стека после вызова процедуры TESTA(i, j, k).

Положение вершины стека	Содержимое стека (длина ячейки 16 бит)	Значение смещения относительно регистра BP
SP	????	
	BP	
	IP	+2
	K	+4
	J	+6
	I	+8

В конце процедуры происходит восстановление значения регистра **BP** и возврат в точку вызова с очисткой стека. В данном случае было занято под параметры 6 байт. Директива **EQU** (**EQUivalent** — Эквивалент) применяется для удобства сопоставления смещения в стеке с реальными параметрами. Вообще обратите внимание на эту директиву — она может существенно повысить читабельность программы и упростить программирование на Ассемблере.

Синтаксис: *Имя* **EQU** *выражение*

Посмотрим теперь на машинный код.

Фрагмент файла листинга ASMPARM.lst

```
10    0000              .MODEL small,pascal
11    0000              .CODE
12                      PUBLIC TESTA
13    0000            TESTA PROC
14    0000 55           push bp
15    0001 8B EC        mov bp,sp
16    0003 8B 46 08     mov ax,[bp+i]   ;get i
17    0006 03 46 06     add ax,[bp+j]   ;add j    to i
18    0009 2B 46 04     sub ax,[bp+k]   ;subtract k from the sum
19    000C 5D           pop bp
20    000D C2 0006      ret 6 ;return, discarding 6 parameter bytes
21    0010            TESTA ENDP
22                      END
```

Здесь появились ***новые режимы адресации***, попробуйте сами с ними разобраться — вы уже вполне сможете! А не получится, не расстраивайтесь — мы этим делом займемся при анализе примера 9.3 (см. п. 9.1.2.4).

Данный вариант реализации предполагает, что программист четко знает, как обрабатываются параметры.

Вариант 2. Современные компиляторы с Ассемблера поддерживают более простой вариант реализации процедур (функций) с параметрами. Рассмотрим теперь его. Здесь будет применена очень удобная директива **ARG**, в которой должны быть перечислены по порядку все формальные параметры, далее указывается имя переменной (в нашем случае **argLen**), в которой подсчитывается количество байт, занимаемых параметрами, и, если нужно, указывается возвращаемое значение (параметр **RETURNS**).

Синтаксис директивы:

ARG *список_формальных_параметров* [*=их_длина*] [**RETURNS** *возв_значение*]

При этом с параметрами и возвращаемым значением можно работать так, как мы привыкли, не заботясь о принятых соглашениях. В конце процедуры (перед возвратом в точку вызова) нужно восстановить из стека содержимое регистра **BP** (это не очень трудно!) и саму команду возврата в точку вызова записать следующим образом:

```
ret их_длина ;argLen
```

Исходный текст модуля ASMPARM.ASM — вариант 2.

```
; Turbo Assembler Copyright (c) 1998 By Голубь Н.Г.
; ASMPARM.ASM
; ——— i+j-k : integer or word ————
; Called as: TESTA(i, j, k);
; Вариант 2 — директива ARG
        .MODEL small,pascal
        .CODE
        PUBLIC TESTA
TESTA   PROC
;Function TESTA( i, j, k:integer):INTEGER;
ARG i:WORD,j:WORD,k:WORD=argLen RETURNS x:WORD
```

```
        mov   ax,i
        add   ax,j        ; i+j
        sub   ax,k        ; i+j-k
        mov   x,ax
        pop   bp
        ret   argLen
TESTA ENDP
        END
```

Посмотрим на файл листинга. В части реализации вычислений с параметрами он точно такой же, как и предыдущий. Но, по-моему, работать с директивой ARG все же удобнее, особенно начинающему программисту.

Фрагмент файла листинга ASMPARM.lst

```
     5                          ; Вариант 2 — директива ARG
     6    0000                          .MODEL small,pascal
     7    0000                          .CODE
     8                                  PUBLIC TESTA
     9    0000                  TESTA PROC
    10                          ;Function TESTA( i, j, k:integer):INTEGER;
    11      =0006 ARG i:WORD,j:WORD,k:WORD =argLen RETURNS x:WORD
1   12    0000 55                       PUSH BP
1   13    0001 8B EC                    MOV BP,SP
1   14    0003 8B 46 08                         mov ax,i
    15    0006 03 46 06                         add ax,j ; i+j
    16    0009 2B 46 04                         sub ax,k ; i+j-k
    17    000C 89 46 0A                         mov x,ax
    18    000F 5D                         pop bp
    19    0010 C2 0006                          ret argLen
    20    0013                  TESTA ENDP
    21                                  END
```

Из файла листинга мы также видим, что директива ARG является еще и МАКРОКОМАНДОЙ (речь об этом впереди!), которая порождает *макрорасширение* (оно в листинге выделено цифрой 1 (слева от номера строки) — это глубина порождаемого кода).

Вариант 3. И, наконец, самый простой вариант, о нем я упоминала в п. 6.1.1.1. Но он подходит только для варианта **Borland Pascal+TASM**. В этом случае в Паскале вы можете описывать и, соответственно, вызывать процедуру на Ассемблере, как хотите (с параметрами или без них, что мы, кстати, до сих пор и делали). Но в этом случае на Ассемблере о стеке забудьте, данные должны быть глобальными и передаваться через сегмент данных. Паскаль — замечательный язык! Спасибо Никлаусу Вирту! Недаром он был придуман им для обучения НАЧИНАЮЩИХ программистов.

9.1.2.4. Соглашения о вызовах в Borland C/C++

Здесь, конечно, все серьезнее (хотя современные компиляторы C/C++ поддерживают вызовы процедур и функций в стиле Паскаль — Windows ведь сначала писали на Паскале). Но мы с вами разберемся с родным для C/C++ стилем вызова функций.

Параметры в стек передаются по порядку, но **справа налево** (ЗАДОМ НАПЕРЕД), т.е. первым в стек помещается ПОСЛЕДНИЙ параметр, последним — ПЕРВЫЙ. По окончании работы **ВЫЗЫВАЮЩАЯ процедура должна очистить стек.** Покажем все это на примере.



ПРИМЕР 9.3.

Возьмем за основу **пример 6.2**, но немного изменим реализацию нашей задачи. Пусть глобальными будут только переменные, связанные с промежуточными вычислениями (числитель — **X1:Dword** и знаменатель — **X2:word**). Все остальные данные будем передавать как параметры. По аналогии с Паскалем реализуем сначала серьезный вариант, где обо всем заботится сам программист, а затем — более простой — через директиву **ARG**.

Исходный текст модуля ASMPARM.ASM — вариант 1.

```
        title Арифметические выражения
; tasm PRIM.ASM /l /ml !!!!!!!!!!
        model large,C
; CopyRight by Голубь Н.Г., 1993-1997, 2000-2001
;x=(2*a + b*c)/(d-a),
; d<>a !!! Проверяется на этапе ВВОДА !!!!
        ;int x,a,b,c,d;
        CODESEG
        Extrn     C X1:Dword
        Extrn     C X2:word ; ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
        Public    C prim
; int prim (int a, int b, int c, int d);
prim    proc      far
        push      bp
        mov       bp,sp ; указатель bp — на вершину стека
;!!!!!!!!!! параметры в стеке хранятся в ОБРАТНОМ
;       порядке
;       CS        EQU [bp+2] ; FAR!!!
;       IP        EQU [bp+4]
a       EQU       [bp+6]
b       EQU       [bp+8]
c       EQU       [bp+10]
d       EQU       [bp+12]
        mov       ax,2
        Imul      a           ; <dx>:<ax>=2*a
        mov       bx,dx       ; bx <=== ст.часть (dx)
        mov       cx,ax       ; cx <=== мл.часть (ax)
        mov       ax,b
        Imul      c           ; <dx>:<ax>=b*c
        add       ax,cx       ; <ax>=<ax>+<cx> (мл.часть)
        adc       dx,bx       ; <dx>=<dx>+<bx> (ст.часть)
        mov       word PTR X1, ax
        mov       word PTR X1+2,dx ; числитель
        mov       cx,d
        sub       cx,a        ; <cx>=<cx>-a
        mov       X2,cx       ; знаменатель
        Idiv      cx          ; <ax>=<dx>:<ax>/<cx>
        ;mov      X,ax        !! Делать НЕ нужно !!!!!!!!
        pop       bp
        ret
prim    endp
        end
```

Попробуем в этом ASM-модуле разобраться. Для этого нам, как и в Паскале, нужно понять процесс передачи фактических параметров, которые должны храниться в стеке. И вообще нужно знать, что находится в стеке на момент выполнения нашей функции.

1. Модель **large** (см. табл. 3.3) означает, что все процедуры в данном модуле имеют атрибут **FAR**.
2. Функцию **int prim (int a, int b, int c, int d)** вызывает C++.
3. Мы уже знаем, что ДО вызова функции в стек должны быть переданы параметры — в данном случае в НЕ естественном для европейца порядке, но в естественном порядке, например, для араба (справа налево: **d,c,b,a**):

```
PUSH d
PUSH c
PUSH b
PUSH a
```

Таким образом, о передаче параметров в данном случае заботится тоже C++.

4. Затем следует ДАЛЬНИЙ вызов функции: **CALL prim**. Это влечет за собой опять загрузку стека, на этот раз адресом возврата в точку вызова:

```
PUSH IP
PUSH CS
```

5. Теперь начинаем выполнять нашу функцию, реализованную на Ассемблере. В самом начале нужно установить какой-то указатель на вершину стека. Для этого, как уже упоминалось в п. 9.1.2.3, используется регистр **BP**:

```
push bp
mov bp,sp
```

Таблица 9.2. Содержимое стека после вызова процедуры prim (int a, int b, int c, int d.

Положение вершины стека	Содержимое стека (длина ячейки 16 бит)	Значение смещения относительно регистра BP
SP	????	
	BP	
	CS	+2
	IP	+4
	a	+6
	b	+8
	c	+10
	d	+12

В конце процедуры происходит восстановление значения регистра **BP** и возврат в точку вызова. Очистка стека здесь НЕ делается, этим должна заниматься программа на C++, поскольку именно она вызвала функцию (процедуру). Директива EQU применяется для удобства сопоставления смещения в стеке с реальными параметрами.

Посмотрим теперь на машинный код.

Фрагмент файла листинга PRIM.lst — вариант 1.

```
7    0000        CODESEG
8                Extrn      C X1:Dword
9                Extrn      C X2:word ; ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
10               Public C prim
11   0000        prim proc  far ;C, a:word, b:word, c:word, d:word
12   0000 55     push bp
```

```
   13   0001 8B EC        mov bp,sp ; указатель bp — на вершину стека
   14      ;!!!!!!! параметры в стеке хранятся в ОБРАТНОМ порядке!!!!
   15                     ;CS   EQU [bp+2]
   16                     ;IP   EQU [bp+4]
   17    = [bp+6]         a     EQU [bp+6]
   18    = [bp+8]         b     EQU [bp+8]
   19    = [bp+10]        c     EQU [bp+10]
   20    = [bp+12]        d     EQU [bp+12]
   21   0003 B8 0002            mov ax,2
   22   0006 F7 6E 06           Imul a      ; <dx>:<ax>=2*a
   23   0009 8B DA              mov bx,dx   ; bx <=== ст.часть    (dx)
   24   000B 8B C8              mov cx,ax   ; cx <=== мл.часть    (ax)
   25   000D 8B 46 08           mov ax,b
   26   0010 F7 6E 0A           Imul c      ; <dx>:<ax>=b*c
   27   0013 03 C1              add ax,cx   ; <ax>=<ax>+<cx> (мл.часть)
   28   0015 13 D3              adc dx,bx   ; <dx>=<dx>+<bx> (ст.часть)
   29   0017 2E: A3 0000e       mov word PTR X1, ax
   30   001B 2E: 89 16 0002e    mov word PTR X1+2,dx ;  числитель
   31   0020 8B 4E 0C           mov cx,d
   32   0023 2B 4E 06           sub cx,a    ; <cx>=<cx>-a
   33   0026 2E: 89 0E 0000e    mov X2,cx   ;        знаменатель
   34   002B F7 F9              Idiv cx     ; <ax>=<dx>:<ax>/<cx>
   35                           ;mov X,ax !!!!!! Делать НЕ нужно !!!!!!!
   36   002D 5D                 pop bp
1  37   002E CB                 RET 00000h
   38   002F              prim  endp
39                        end
```

Давайте мы ради любопытства проанализируем какой-нибудь машинный код, поскольку подпрограмма на Ассемблере внешне выглядит, как обычно мы писали подпрограмму без параметров. Эта хитрость получилась за счет директивы **EQU**. Например, возьмем следующую команду:

```
25  000D 8B 46 08        mov ax,b
```

Вспомним анализ примера в п. 8.2 (строка 21). Принцип один и тот же, только режим адресации здесь другой. Итак, **КОП = 8B**. Он соответствует команде **MOV r16,r/m** (см. Табл. П7.1). Это обычная двухадресная команда, у которой есть байт способа адресации (БСА). Распишем КОП и БСА в двоичной системе счисления: 1000 1011 0100 0110 и попробуем их расшифровать. Это — однозначно вариант 1 (см. рис. 8.2 и 8.7):

	Код операции							
Биты	7	6	5	4	3	2	1	0
	1	0	0	0	1	0	1	1
	КОП						d	w

	Байт способа адресации							
Биты	7	6	5	4	3	2	1	0
	0	1	0	0	0	1	1	0
	mod		reg			r/m		

Проанализируем БСА совместно с КОП. Из табл. 8.3 получаем: **mod=01**, значит, **r/m=110=BP+смещение** (см. табл. 8.3). Смещение длиной в 1 байт должно находиться в самом машинном коде — это **08**. Таким образом, получаем **BP+08** — это наш формальный параметр **b** (режим адресации *по базе BP со смещением*). Значение собственно поля **reg=000=AX** (**w=1**, см. табл. 8.1). Бит **d=1**, значит, регистр **AX** является приемником. Как видите, тоже ничего страшного...

Теперь перейдем к варианту с использованием директивы-макрокоманды **ARG**.

Исходный текст модуля ASMPARM.ASM — вариант 2.

```
        title Арифметические выражения
        ; tasm PRIM.ASM /l /ml !!!!!!!!!!
        model large,C
; CopyRight by Голубь Н.Г., 1993-1997, 2000-2001
;x=(2*a + b*c)/(d-a), d<>a !!!
        ;int x,a,b,c,d;
        CODESEG
        Extrn     C X1:Dword
        Extrn     C X2:word ; ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
        Public    C prim
prim    proc      far
;ARG i:WORD,j:WORD,k:WORD =argLen RETURNS x:WORD
ARG a:word, b:word, c:word, d:word RETURNS X:WORD
        mov       ax,2
        Imul      a                     ; <dx>:<ax>=2*a
        mov       bx,dx                 ; bx <=== ст.часть (dx)
        mov       cx,ax                 ; cx <=== мл.часть (ax)
        mov       ax,b
        Imul      c                     ; <dx>:<ax>=b*c
        add       ax,cx                 ; <ax>=<ax>+<cx> (мл.часть)
        adc       dx,bx                 ; <dx>=<dx>+<bx> (ст.часть)
        mov       word PTR X1, ax
        mov       word PTR X1+2,dx      ; числитель
        mov       cx,d
        sub       cx,a                  ; <cx>=<cx>-a
        mov       X2,cx                 ; знаменатель
        Idiv      cx                    ; <ax>=<dx>:<ax>/<cx>
        mov       X,ax
        ;pop      bp - в C++ НЕЛЬЗЯ!!!!!!!!!!!!!!!!!
        ret
prim    endp
        end
```

Посмотрим теперь на машинный код. **Обратите внимание** на разницу в реализации компилятором директивы **ARG** по сравнению с вариантом, предназначенным для стыковки с Паскалем!

Фрагмент файла листинга PRIM.lst — вариант 2.

```
   7 0000                     CODESEG
   8                          Extrn C X1:Dword
   9                          Extrn C X2:word ; ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
   10                         Public C prim
   11 0000               prim proc far
   12  ;ARG i:WORD,j:WORD,k:WORD =argLen RETURNS x:WORD - PASCAL!!!!
   13               ARG a:word, b:word, c:word, d:word RETURNS X:WORD
1  14 0000 55                 PUSH BP
1  15 0001 8B EC              MOV BP,SP
1  16 0003 B8 00 02           mov ax,2
   17 0006 F7 6E 06           Imul a     ; <dx>:<ax>=2*a
   18 0009 8B DA              mov bx,dx ; bx <=== ст.часть (dx)
   19 000B 8B C8              mov cx,ax ; cx <=== мл.часть (ax)
   20 000D 8B 46 08           mov ax,b
   21 0010 F7 6E 0A           Imul c     ; <dx>:<ax>=b*c
   22 0013 03 C1              add ax,cx ; <ax>=<ax>+<cx> (мл.часть)
```

```
    23  0015 13 D3              adc dx,bx ; <dx>=<dx>+<bx> (ст.часть)
    24  0017 2E A3      0000e   mov word PTR X1, ax
    25  001B 2E: 89 16  0002e   mov word PTR X1+2,dx ;   числитель
    26  0020 8B 4E 0C           mov cx,d
    27  0023 2B 4E 06           sub cx,a  ; <cx>=<cx>-a
    28  0026 2E: 89 0E  0000e   mov X2,cx ;         знаменатель
    29  002B F7 F9              Idiv cx   ; <ax>=<dx>:<ax>/<cx>
    30  002D 89 46 0E           mov X,ax
    31                          ;pop bp - в C++ НЕЛЬЗЯ!!!!!!!!
1   32  0030 5D                 POP BP
1   33  0031 CB                 RET 00000h
    34  0032            prim    endp
    35                          end
```

Реализации варианта 1 и 2 в части обработки формальных параметров получились одинаковыми. В варианте 2 можно НЕ думать о том, в каком регистре должен храниться возвращаемый результат работы функции, — для этого есть параметр **RETURNS**.

9.2. Команды условной передачи управления Jcc

Для процессоров i8086/i286 все команды условного перехода реализуют короткий переход (**SHORT**), т.е. смещение в пределах [-128...127]. Если вас по какой-то причине это не устраивает, нужно воспользоваться двумя командами **Jcc** и **JMP**. Или воспользоваться 32-разрядным Ассемблером (речь о нем впереди).

Базовых команд условного перехода всего 17, но они могут иметь различную мнемонику (это команды-синонимы — для удобства чтения и понимания программы), поэтому получается 31 команда. Читать эти команды достаточно просто, если знаешь ключ.

Первая буква команды **J** от уже известного нам слова (Jump — прыжок). Остальные буквы (*cc*) в сокращенном виде описывают условие перехода. По этому признаку все команды условного перехода можно разбить на три группы.

Первая группа команд условного перехода.

1. **E** — Equal (равно).
2. **N** — Not (не, отрицание).
3. **G** — Greater (больше) — применяется для чисел со ЗНАКОМ.
4. **L** — Less (меньше) — применяется для чисел со ЗНАКОМ.
5. **A** — Above (выше, больше) — применяется для чисел БЕЗ ЗНАКА.
6. **B** — Below (ниже, меньше) — применяется для чисел БЕЗ ЗНАКА.

Например, теперь понятно, что может означать команда **JL** — переход, если меньше. Ей эквивалентна команда-синоним **JNGE** — переход, если НЕ больше и НЕ равно. Разница в командах перехода для знаковых и беззнаковых данных объясняется тем, что они реагируют на РАЗНЫЕ флаги (для знаковых данных существенен флаг **SF**, а для беззнаковых — **CF**) — см. табл. 9.3.

Условный переход для этой группы команд обычно реализуется в два шага:

- **Сравнение**, в результате чего формируются флаги (регистр **FLAGS** — см. п. 3.3.4).
- **Условная передача управления** (**J***cc* *Короткая_метка*) на помеченную команду в зависимости от значения флагов.

Таким образом, в данном случае реализуется условный оператор **if**.

Эта группа операций чаще всего выполняется в паре с командой сравнения **CMP** (п. 5.2.2):

CMP *приемник, источник*

J*cc* *Короткая_метка*

О состоянии флагов после выполнения команды **CMP** см. табл. 5.8.

За состоянием флагов очень удобно наблюдать в отладчиках.

Таблица 9.3. Команды условного перехода группы 1.

Условие сравнения (CMP)	Мнемокод/синоним	Состояние флагов
Для любых чисел и кодов		
Приемник = источник	**JE**	**ZF=1**
Приемник <> источник	**JNE**	**ZF=0**
Для чисел со ЗНАКОМ		
Приемник < источник	**JL/JNGE**	**SF<>OF**
Приемник <= источник	**JLE/JNG**	**SF<>OF or ZF=1**
Приемник > источник	**JG/JNLE**	**SF=OF and ZF=0**
Приемник >= источник	**JGE/JNL**	**SF=OF**
Для чисел БЕЗ знака		
Приемник < источник	**JB/JNAE**	**CF=1**
Приемник <= источник	**JBE/JNA**	**CF=1 or ZF=1**
Приемник > источник	**JA/JNBE**	**CF=0 and ZF=0**
Приемник >= источник	**JAE/JNB**	**CF=0**

ПРИМЕР 9.4.

Промоделируем на Ассемблере простейшую задачу для 16-разрядных знаковых и беззнаковых данных:

```
int a, b;                          unsigned int c, d;
if (a=b) then Fsign = 0;           if (c=d) then Fusign = 0;
if (a<b) then Fsign = -1;          if (c<d) then Fusign = -1;
if (a>b) then Fsign = 1;           if (c>d) then Fusign = 1;
```

Пусть глобальными будут переменные Fsign и Fusign (тип int). Все остальные данные будем передавать как параметры. Реализуем решение нашей задачи через директиву **ARG** в интегрированной среде программирования **Borland C++ 5.02+TASM-5**.

Исходный текст модуля PRIMIF.ASM

```
title Условные переходы — условный оператор if
   ; tasm PRIMIF.ASM /l /ml !!!!!!!!!
   model large,C
   ; CopyRight by Голубь Н.Г., 2001
   ;int a,b;
   ;UNSIGNED int c,d;
   CODESEG
   EXTRN C Fsign:WORD,Fusign:WORD ; ГЛОБАЛЬНЫЕ
   Public  C primIf
primIf proc far
ARG a:word, b:word, c:word, d:word
```

```
; ———— ЗНАКОВЫЕ данные
        mov   Fsign, 0   ; a=b
        mov   ax,a
        mov   bx,b
        cmp   ax,bx
        JL    Less       ; a<b
        JG    Great      ; a>b
        jmp   Cont
Less:   mov   Fsign,-1   ; a<b
        jmp   Cont       ; a>b
Great:  mov   Fsign, 1
; ———— БЕЗЗНАКОВЫЕ данные
Cont:   mov   Fusign, 0 ; c=d
        mov   ax,c
        mov   bx,d
        cmp   ax,bx
        JB    Below      ; c<d
        JA    Above      ; c>d
        jmp   Exit
Below:  mov   Fusign,-1 ; c<d
        jmp   Exit
Above:  mov   Fusign, 1 ; c>d
Exit:   ret
primIf  endp
        end
```

Можете самостоятельно поупражняться в чтении машинных кодов (анализ файла Primif.lst).

Вторая группа команд условного перехода.

Эта группа команд реагирует на то или иное значение ОПРЕДЕЛЕННОГО ФЛАГА. Поэтому в мнемонике данной группы команд всегда указывается первая буква проверяемого флага. Эти команды НЕ требуют ОБЯЗАТЕЛЬНОГО наличия команд сравнения перед своим выполнением. Им достаточно ЛЮБОЙ команды, которая вырабатывает нужный флаг.

Таблица 9.4. Команды условного перехода группы 2.

Мнемокод/синонимы	Состояние флагов	Мнемокод/синонимы	Состояние флагов
JZ/JE	ZF=1	JNZ/JNE	ZF=0
JS	SF=1	JNS	SF=0
JC/JB/JNAE	CF=1	JNC/JAE/JNB	CF=0
JO	OF=1	JNO	OF=0
JP	PF=1	JNP	PF=0

Как видно из таблицы, с некоторыми командами мы уже встречались в группе 1. Синонимы мнемокодов команд просто подчеркивают, на что конкретно обратил внимание программист.

Например, проверим на переполнение результат операции умножения двух беззнаковых 8-разрядных данных **a** и **b**:

```
    MOV   AL,a
    MUL   b
    JC    ERRORmul  ;  a*b > 255  (CF=1)
          ..............
ERRORmul:
          ..............
```

Применение в данном контексте программы мнемоники команд-синонимов **JB** или **JNAE** НЕ даст того понимания ситуации с вычислениями, которое было вложено в команду **JC**.

Третья группа команд условного перехода.

В эту группу входит всего одна команда **JCXZ**. Она, в отличие от предыдущих команд, проверяет НЕ флаги (к ним она БЕЗРАЗЛИЧНА!), а содержимое регистра **CX** на ноль (Jump if CX is Zero).

Синтаксис:

JCXZ *Короткая_метка*

Логика работы команды:

If (CX=0) then goto *Короткая_метка*

Это ОЧЕНЬ важная команда, которая используется при организации циклов.

9.3. Команды управления циклами LOOPx

Команды этого вида организуют циклические вычисления (LOOP — цикл), используя регистр счетчика **CX** по своему прямому назначению (*counter — счетчик*). В регистр **CX** должно быть предварительно занесено количество повторений цикла. Эти команды реализуют классический *цикл со счетчиком с постусловием*.

9.3.1. Команда LOOP — переход по счетчику

Синтаксис команды: **LOOP** *короткая_метка*

Логика работы команды:

<CX> = Counter

short_label: Выполнение тела цикла

<CX> = <CX> – 1

if (<CX> <> 0) goto short_label

Аналог реализации команды LOOP на Ассемблере:

```
              MOV CX, Counter
short_label:
          ; Выполнение тела цикла
          ; Проверка условия ПРОДОЛЖЕНИЯ цикла
              DEC CX
              CMP CX, 0
              JNE short_label
```

Команда **LOOP** уменьшает содержимое регистра **CX** на 1, затем передает управление метке short_label, если содержимое **CX** не равно 0. Передача управления на метку short_label для базовых процессоров — только КОРОТКАЯ [-128,0]. Поскольку условие выхода из цикла проверяется в КОНЦЕ, при значении Counter=0 цикл все равно выполнится. Этого мало, **мы еще и зациклимся** (почему?). Чтобы этого избежать, обычно ДО НАЧАЛА ЦИКЛА проверяют содержимое регистра **CX** на ноль. Таким образом, стандартная последовательность команд для организации цикла СО СЧЕТЧИКОМ имеет следующий вид:

```
                      MOV  CX, Counter
                      JCXZ ExitCicle ; если <CX> = 0, цикл ОБОЙТИ
   short_label:
                    ; Выполнение тела цикла
                      LOOP short_label
```

ПРИМЕР 9.5.

Вычислить значение факториала **p** = **n!** = **1*2*3*...*n**. Известно, что **0!** = **1**. Отрицательным значение **n** быть НЕ может (по правилам математики). Исходный код asm-модуля может иметь следующий вид:

Исходный текст модуля Fact.asm

```
     Model Large,C
; определение префикса для локальных меток
     locals @@
     codeseg
     Extrn C n:Word
     Extrn C p:Word
     Public proizv1
Proizv1 Proc far
; Вариант 1
     mov cx,n        ; количество повторений
     mov si,1
     mov ax,si
     jcxz @@Exit     ;if cx=0 then Exit
@@begin: ;======= НАЧАЛО цикла ==========
     mul si          ; <dx:ax> = <ax>*si
     inc si
     ;==== Выход из цикла =================
     loop @@begin
@@Exit:
     mov p,ax
     ret
proizv1 endp
;-----------------------
     Public proizv2
Proizv2 Proc far
; Вариант 2
     mov cx,n        ; количество повторений
     mov ax,1
     jcxz @@Exit     ;if cx=0 then Exit
@@begin: ;======= НАЧАЛО цикла ==========
     mul cx ; <dx:ax> = <ax>*cx
     ;==== Выход из цикла =================
     loop @@begin
@@Exit:
     mov p,ax
     ret
proizv2 endp
end
```

Вариант 2 позволит нам сэкономить 3 байта за счет использования регистра CX одновременно как регистр-счетчик и как индекс. В реальных задачах такая экономия получается НЕ всегда (и НЕ стоит ею увлекаться!).



ОБРАТИТЕ ВНИМАНИЕ

на **директиву locals @@**. Она позволяет нам НЕ думать о дублировании имен меток в разных подпрограммах. Метки с префиксом @@ считаются локальными. Если компилятор встретит метку с таким же именем, он просто при компиляции присвоит ей другое имя (обычно эти метки получают в конце имени номер, который увеличивается на единицу — все очень просто!).

Напишите главную программу на каком-либо алгоритмическом языке и проверьте работоспособность данного модуля.

Вопросы

1. Для стыковки с программой на каком алгоритмическом языке предназначен данный asm-модуль?
2. Почему в исходном коде используется БЕЗЗНАКОВОЕ умножение?
3. Можно ли использовать знаковое умножение?
4. Что нам дает применение БЕЗЗНАКОВОГО умножения?
5. При каком предельном значении **n** получается верный результат?

9.3.2. Команда LOOPE (LOOPZ) переход по счетчику и если равно

Данная команда имеет два равнозначных мнемонических имени (if Equal — если Равно или if Zero — если Ноль). На мой взгляд, мнемоника команды **LOOPE** — более человеческая, а **LOOPZ** — более машинная, т.к. предполагает знание того факта, что если два операнда равны, флаг нуля **ZF=1** (*установлен*).

Синтаксис команды: **LOOPE** короткая_метка
LOOPZ короткая_метка

Логика работы команды:

```
                  <CX> = Counter
short_label: Выполнение тела цикла
                  <CX> = <CX> — 1
                  if (<CX> <> 0 and <ZF> = 1) goto short_label
```

Все то, что говорилось для команды **LOOP**, справедливо и для команды **LOOPE (LOOPZ)**, добавляется еще проверка флага **ZF**. Применяется данная команда в случае, если нужно досрочно выйти из цикла, как только находится ПЕРВЫЙ элемент, ОТЛИЧНЫЙ от заданной величины.

9.3.3. Команда LOOPNE (LOOPNZ) переход по счетчику и если НЕ равно

Данная команда тоже имеет два равнозначных мнемонических имени (if Not Equal — если НЕ равно или if Not Zero — если НЕ ноль). В отличие от предыдущей команды проверяется, *сброшен* ли флаг нуля ZF=0.

Синтаксис команды: **LOOPNE** короткая_метка
LOOPNZ короткая_метка

Логика работы команды:

<CX> = Counter

short_label: Выполнение тела цикла

<CX> = <CX> — 1

if (<CX> <> 0 and <ZF> = 0) goto short_label

Все то, что говорилось для предыдущей команды, справедливо и для команды **LOOPNE (LOOPNZ)**. Применяется данная команда в случае, если нужно досрочно выйти из цикла, как только находится ПЕРВЫЙ элемент, РАВНЫЙ заданной величине.



ПРИМЕР 9.6.

Вычислить значение суммы чисел натурального ряда: **s = 1+2+3+...+n**. Вычисления закончить, как только значение суммы станет равным некоторому числу **k** или не будут перебраны все **n** чисел. Исходный код модуля может иметь следующий вид:

Исходный текст модуля Sum.asm

```
; sum.asm
     Model      Large,C
     locals     @@
     codeseg
     Extrn      C n:Word
     Extrn      C s:Word
     Extrn      C k:Word
     Public     sum
sum Proc far
     mov        cx,n        ; количество повторений
     xor        ax,ax
     xor        si,si
     jcxz       @@Exit      ;if cx=0 then Exit
@@begin:;=======НАЧАЛО цикла =============
     inc        si
     add        ax,si
     cmp        ax,k
;====Выход из цикла, если <ax>=k или <cx>=0
     loopNE @@begin
@@Exit:
     mov s,ax
       ret
sum endp
     end
```

Исходный текст программы sum_cpp.cpp

```
/* sum_cpp.cpp Borland C++ 5.02
 Вычислить значение s=1+2+3+...+i+...+n.
 Вычисления закончить, как только sum=k
       (c) Copyright 2001 by Голубь Н.Г.
*/

#include <conio.h>
#include <fstream.h>
#include "convert.h"

// Глобальные переменные для передачи в asm-модуль
int      n,k;
unsigned int s;
const char* TITLE =
       TXT("\nВычислить s=1+2+3+...+n\nВычисления закончить, как только sum=k\n");
const char* INVITE = TXT("Введите значение ");
const char* QUEST = TXT("Хотите ли Вы продолжить вычисления? Или закончить их?");

// Подключение внешнего модуля
extern "C"
  { void far sum(void); }
// Проверка на допустимый диапазон [-32768..32767]
int test( long int a)
  {      return((a>>15)+1)&~1; };
template <class anyType>
int input (anyType& n)
  { ifstream my_inp ("CON");
    ofstream my_out ("CON");
   my_inp >> n;
    switch (my_inp.rdstate())
        {case ios::goodbit: return 0;
        case ios::eofbit : return 0;
        case ios::failbit:
        case ios::badbit :
          my_out << "\n!!!!! ERROR !!!!!";
           my_out << "\nLet's try once again.....\n";
          return 1;
          }
  }
unsigned long sum_cpp(int n, int k)
    { unsigned long s=0;
      for (int i=1;i<=n;i++)
        { s += i;
          if (s==k) return s;
        }
     return s;
    }
int main(void)
    { char flag='y';
     long temp;
     int j=1;
     do
        { clrscr();
        cout << "\============ TEST #" << j++ << "=================\n";
         cout << TITLE;
```

```
    do
       { cout << INVITE << "n=[0..32767]: ";
         while(input(temp));
       } while (temp <0 || test(temp));
 n=(int)temp; // Присваивание, если все нормально с допустимым диапазоном
 do
       { cout << INVITE<< "k=[0..32767]: ";
         while(input(temp));
       } while ( test(temp));
 k= (int)temp;
 unsigned long ff = sum_cpp(n,k);
 unsigned int f = (unsigned int)ff;
 // Проверка результата на превышение ДОПУСТИМОГО диапазона LONG INT
 if (f == ff)
   { cout <<"Result(C++):" << f <<endl;
     sum();
     cout <<"Result(ASM):" << s <<endl;
   }
 else
     cout << "\n!!!! The Result " << ff << " has exceeded a VALID range!!!!\n";
    cout << QUEST <<" <y/CTRL+C>";
     cin   >>flag;
   } while(flag=='y');
 return 0;
}
```

Результат вычисления имеет следующий вид:

```
   ============= TEST #1==================
   Вычислить s=1+2+3+...+n
   Вычисления закончить, как только sum=k
   Введите значение n=[0..32767]: 23
   Введите значение k=[0..32767]: 23324
   Result(C++):276
   Result(ASM):276
Хотите ли Вы продолжить вычисления? Или закончить их? <y/CTRL+C>
   ============= TEST #2==================
   Вычислить s=1+2+3+...+n
   Вычисления закончить, как только sum=k
   Введите значение n=[0..32767]: 23
   Введите значение k=[-32768..32767]: 231
   Result(C++):231
   Result(ASM):231
Хотите ли Вы продолжить вычисления? Или закончить их? <y/CTRL+C>
   ============= TEST #3==================
   Вычислить s=1+2+3+...+n
   Вычисления закончить, как только sum=k
   Введите значение n=[0..32767]: 3333
   Введите значение k=[0..32767]: 333
   !!!! The Result 5556111 has exceeded a VALID range!!!!
Хотите ли Вы продолжить вычисления? Или закончить их? <y/CTRL+C>
   ============= TEST #4==================
   Вычислить s=1+2+3+...+n
   Вычисления закончить, как только sum=k
   Введите значение n=[0..32767]: 4444
```

```
   Введите значение k=[0..32767]: 444
   !!!! The Result 9876790 has exceeded a VALID range!!!!
Хотите ли Вы продолжить вычисления? Или закончить их? <y/CTRL+C>

   ============ TEST #5================
   Вычислить s=1+2+3+...+n
   Вычисления закончить, как только sum=k
   Введите значение n=[0..32767]: 5
   Введите значение k=[0..32767]: 10
   Result(C++):10
   Result(ASM):10
Хотите ли Вы продолжить вычисления? Или закончить их? <y/CTRL+C>
```

9.4. Основные принципы организации и обработки массивов

Вы, конечно, знаете, что *массив — это упорядоченная последовательность ОДНОТИПНЫХ данных, которые хранятся в оперативной памяти.*

Поскольку язык С/С++ является наиболее близким к Ассемблеру, проиллюстрируем основные идеи организации и обработки массивов с привлечением языка С/С++.

9.4.1. Одномерные массивы

Пусть два массива и указатели на них в языке С/С++ описаны следующим образом:

int ArrI [10], *PtI; PtI=ArrI;

float ArrF [5], *PtF; PtF=ArrF;

Изобразим графически основные характеристики этих массивов.

Байты в ОЗУ	1	2	3	4	...	17	18	19	20
Элементы массива (значения)	ArrI[0]		ArrI[1]		...	ArrI[8]		ArrI[9]	
Указатели (адреса)	ArrI		ArrI+1		...			ArrI+9	
	PtI		PtI+1		...	PtI+8		PtI+9	

РИС. 9.1. *Основные характеристики целочисленного массива int ArrI [10] в языке С/С++*

Байты в ОЗУ	1	2	3	4	5	6	7	8	...	17	18	19	20
Элементы массива (значения)	ArrF[0]				ArrF[1]				...	ArrF[4]			
Указатели (адреса)	ArrF				ArrF+1				...	ArrF+4			
	PtF				PtF+1				...	PtF+4			

РИС. 9.2. *Основные характеристики вещественного массива float ArrF [5] в языке С/С++.*

Напомним, что **PtF + 2** означает то же самое, что **&ArrF[2]** и **ArrF+2**, — это один и тот же *адрес* элемента массива **ArrF[2]**. Теперь это четко видно и на рис. 9.2. А ***(PtF + 2)** означает то же самое, что **ArrF[2]** и ***(ArrF+2)**, — это одно и то же *значение* элемента массива **ArrF[2]**. **Это — самый скользкий момент в языке С**, и не одно поколение программистов на этом спотыкалось. Если вы этот момент поймете, массивы, указатели в любом языке программирования станут для вас, как родные.

Для того чтобы обрабатывать массив в Ассемблере, нужно знать, где он хранится (его начальный адрес), и длину его элемента. Как и в языке С/С++, имя массива в Ассемблере является также и его начальным адресом.

Режим адресации с индексацией вида *имя_массива*[*регистр_индекс*] позволяет обрабатывать каждый элемент массива. В качестве *регистра_индекса* можно брать любой допустимый для косвенной адресации регистр, например, регистр **BX** — см. табл. 8.3 (регистр **BP** лучше не трогать и вы знаете, почему — см. п. 3.3.7 и п. 9.1.2.3).

Мы можем эти массивы передавать как параметры или как внешние имена, например, следующим образом:

```
EXTRN C ArrI:WORD, ArrF:DWORD
```

Байты в ОЗУ	1	2	3	4	5	6	7	8	...	17	18	19	20
Индекс-переменная (i)	0				1				...	4			
Элементы массива (значения)	ArrF[0]				ArrF[1]				...	ArrF[4]			
Адреса (смещения)	ArrF	ArrF +1	ArrF +2	ArrF +3	...				...	ArrF +16	ArrF +17	ArrF +18	ArrF +19
Индекс-регистр (смещение)	BX_0				BX_0+4				...	BX_0+4*i			

РИС. 9.3. *Основные характеристики вещественного массива float ArrF [5] в Ассемблере.*

Основные характеристики этих массивов в Ассемблере будут немного отличаться от рассмотренных нами для С/С++.

В общем случае, если взять в качестве индекс-регистра регистр **BX**, доступ к любому элементу одномерного массива **Array [i]** длины **Larray** подчиняется в Ассемблере следующей закономерности:

$$\mathbf{Array[i]} \rightarrow \mathbf{Array+BX_i = Array[BX_i]},$$

где

$$\mathbf{BX_i = BX_0 + Larray*i = BX_{i-1} + Larray;}$$

BX_0 =0; i=0,...,n-1; n — длина массива Array.

Реализуем все эти знания на конкретном примере.

ПРИМЕР 9.7.

Посчитать сумму всех элементов 16-разрядного целочисленного знакового одномерного массива **A** длиной не более 3000 элементов.

Реализуем вычисления в среде **Borland C++** и **TASM**. Поскольку элементов массива может быть достаточно много, для тестирования нашей задачи применим генерацию элементов массива через датчик случайных чисел, если число элементов массива будет превышать 9, в противном случае ввод элементов массива сделаем вручную.

Исходный текст программы summP.cpp

```
/* summP.cpp Borland C++ 5.02
 Проект !!!!!! DOS (standard) !!!!!! asm+cpp файлы
        (c) Copyright 1998-2000, 2001 by Голубь Н.Г.
Генерация значений элементов массива через датчик случайных чисел, если n>=10.
 Проверка введенных данных на допустимый диапазон.
        Вычислить сумму элементов массива A
*/

#include <iomanip.h> // манипулятор setw(16)
#include <stdlib.h> // int random(int num);
#include <time.h> // void randomize(void);

const int N=3000;
int A[N];
int n;
long S;
extern "C"
   {void SummA (void);}
void inputN(const int N, int& n)
{ do{cout << "\n??? n [1.."<< N <<"]=======>";
        cin >> n;
        cout << "inputN: n = " << n << endl;
   } while (n<1 || n>N);
}

void inputA(int A[], int n)
{ randomize();
  int j=1;
  long temp;
  for (int i=0; i<n; i++)
        if (n<10)
                { cout << "??? A["<< i <<"]=======>";
                  cin >> temp;
                  A[i] = (int)temp;
                if (A[i]!=temp)i—; // !!!!!!!! проверка на диапазон !!!!!!!!!!!!!!!
                }
        else
          { j = -j;
                A[i]= j*(int)random (30000);
                }
 }

void outputA(const int A[], int n)
{ cout << "\n==================================================================\n"
                << " Array: \n";
 for (int i=0; i<n; i++)
        cout << setw(8) << A[i];
 cout << endl;
 }
double summaP(const int A[], int n)
{ double sum=0;
  for (int i=0; i<n; i++)
        sum += (double)A[i];
  return sum;
}
```

```
void main()
{
  for (;;)
      { // ввод реального количества элементов массива A
       inputN(N,n);
       // ввод элементов массива A
       inputA(A,n);
       outputA(A,n);
       cout << "\nThe sum of all elements of a Array\n";
       cout << "C++: " << summaP(A,n) << endl;
       SummA();
       cout << "ASM: " << S;
       cout << "\n Exit Ctrl-C";
      }
}
```

Исходный текст модуля SummA.asm

```
      model large,C
      ; CopyRight by Голубь Н.Г., 2001
      LOCALS     @@
      .CODE
      EXTRN      C A:Word, n:Word, S:DWord
      PUBLIC     C SummA
; Синонимы:
a     EQU        A[BX]
s0    EQU        WORD PTR S       ;мл. часть суммы
s2    EQU        WORD PTR S+2     ;ст. часть суммы
SummA Proc       C far
      xor        bx,bx            ;смещение <BX>=0
      XOR        SI,SI            ;мл. часть суммы
      XOR        DI,DI            ;ст. часть суммы
      mov        cx,n
      jcxz       @@4
@@1:
      mov ax,a
      CWD
      add        SI,ax            ;длинное сложение
      adc        DI,dx
; переход к след. элементу массива A
      inc bx
      inc bx                      ;изменение смещения <BX>=<BX>+2
      LOOP @@1
@@4:
      mov        s0,SI
      mov        s2,DI
      ret
SummA EndP
      End
```

Тестовый пример:

```
??? n [1..3000]=======>33
inputN: n = 33
=================================================================
 Array:
-29843 10693 -27973 24158 -12162 4505  -715   14756 -23280  25979
-18342 1973  -741   18850 -1871  13624 -10892 17217 -500    23614
-9300  27231 -28352 23507 -5766  22057 -24080 6273  -14235  26656
-11167 26239 -22926
The sum of all elements of a Array
C++: 45187
ASM: 45187
Exit Ctrl-C
??? n [1..3000]=======>
```

Полюбопытствуем, что происходит с машинными кодами при обработке массива (файл **Summa.lst** находится в прилагаемых к книге материалах):

7 = A[BX] a EQU A[BX]

17 000D 2E: 8B 87 0000e mov ax,a

Эта команда нам встречалась неоднократно, только *режим адресации* здесь другой (поэтому отличается БСА). Кроме того, здесь есть префикс **2E**, который означает, что данное находится в сегменте кода (см. п. 8.2). Итак, **КОП = 8B**. Он соответствует команде **MOV r16,r/m** (см. табл. П7.1). Это обычная двухадресная команда, у которой есть байт способа адресации (БСА). Распишем КОП и БСА в двоичной системе счисления: 1000 1011 1000 0111 и попробуем их расшифровать. Это — однозначно вариант 1 (см. рис. 8.2 и 8.7):

	Код операции										**Байт способа адресации**							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	1	0	0	0	1	0	1	1			1	0	0	0	0	1	1	1
	КОП						d	W			mod		Reg			r/m		

Проанализируем БСА совместно с КОП. Из табл. 8.3 получаем: **mod=10**, значит, **r/m=111=BX+смещение** (см. табл. 8.3). Смещение длиной в 2 байта должно находиться в самом машинном коде — это **0000e**. Таким образом, получаем **BX+0000e** — это наш внешний элемент массива **A[BX]** (режим адресации *по базе BX со смещением*). Значение собственно поля **reg=000=AX** (**w=1**, см. табл. 8.1). Бит **d=1**, значит, регистр **AX** является приемником.

Ну, а теперь посмотрим на следующие команды:

```
16 000D             @@1:
24 0019  E2  F2          LOOP  @@1
25 001B             @@4:
```

Команда **LOOP** (**КОП = E2** — см. табл. П7.1), как все команды перехода, изменяет содержимое регистра **IP**:

```
  001B
+   F2
------
  000D
```

<IP>	=	<IP>	+	*смещение_к_нужной_команде*
000D	=	001B	+	F2

<IP>=**000D**, что и требовалось получить.

9.4.2. Двухмерные массивы

Для двухмерных массивов (матриц) идея будет та же самая, только нужно определиться, как такой массив будет располагаться в оперативной памяти: по строкам или по столбцам (память-то линейная!). Соответственно, и индекс-регистров тоже будет два. А также — два цикла: внешний и внутренний. Значит, и вычислений прибавится.

Например, пусть имеется некая матрица **Matr[M][N]** и в памяти она располагается по строкам: сначала N элементов первой строки, потом N элементов второй строки и т.д. до строки M. Длину элемента этой матрицы тоже обозначим через **Larray**.

Тогда адрес элемента **Matr[i,j]** будет равен **Matr+N*i* Larray+j**, где **i=0,...,M-1; j=0,...,N-1.** Выделим в Ассемблере для хранения величины **N*i* Larray** регистр **BX**, а для **j** — регистр **SI** (или **DI**). Тогда **Matr[BX]** будет означать начальный адрес строки **i**, а **Matr[BX][SI]** (**Matr[BX+SI]** или **Matr+[BX+SI]** — эти три записи равнозначны) — адрес элемента **j** в этой строке, т.е.

Matr[i,j] → **Matr[BX][SI]**

ПРИМЕР 9.8.

Решим предыдущую задачу, но для матрицы (двухмерного массива) **A**, состоящей из **M<=10 строк** и **N<=5 столбцов**.

Реализуем вычисления в среде **Borland C++** и **TASM**. В C/C++ матрица хранится по строкам. Как вычислять сумму всех элементов такой матрицы (по строкам или по столбцам) алгоритмически безразлично. Поэтому в C/C++ мы будем находить сумму по столбцам (см. файл **MATRIX.cpp** в материалах, прилагаемых к данной книге), а в Ассемблере (для разнообразия и так проще!) — по строкам. Значит, **на Ассемблере внутренний цикл алгоритмически ничем не будет отличаться от предыдущей задачи.** Можем даже взять тот же индекс-регистр! Но поскольку мы на примере иллюстрируем изложенную выше теорию, возьмем регистр **SI**, чтобы не запутаться...



ЗАМЕЧАНИЕ

При выполнении этой задачи на Ассемблере нужно ОЧЕНЬ ХОРОШО знать, как реализуется команда циклического перехода, и НЕ потерять регистр **CX**. Кроме того, необходимо знать максимально возможное число столбцов **N**. В нашем случае **N=5**.

Поясним решение нашей задачи еще и графически.

Байты	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	97	98	99	100
Столбцы (j)	0		1		2		3		4		0		1		...	3		4	
Индекс-регистр SI	0		+2		+4		+6		+8		0		+2		...	+6		+8	
Строки (i)	0										1				I	9			
Индекс-регистр BX	0										+10				+(10*I)	+(10*9)=+90			

РИС. 9.4. *Основные характеристики целочисленной матрицы A [10][5] в Ассемблере.*

Исходный текст модуля SummA2.asm

```
        model  large,C
        ; CopyRight by Голубь Н.Г., 2001
        ; Сумма всех элементов матрицы A[][N]
        LOCALS    @@
        .CODE
        EXTRN     C A:Word, m:Word, n:Word, S:DWord
N       dw   10 ; 2(длина int)*N=2*5 !!!!!!!!!!!
        PUBLIC C  SummA2
a       EQU       A[BX][SI] ;   ДВОЙНАЯ ИНДЕКСАЦИЯ!!!
s0      EQU       WORD PTR S    ;мл. часть суммы
s2      EQU       WORD PTR S+2  ;ст. часть суммы
SummA2  Proc      C far
        xor       bx,bx     ; смещение <BX>=0 (по строкам)
        mov       s0,0
        mov       s2,0
; цикл по строкам
; внешний)==============================
        mov       cx,m
        jcxz      @@4
@@1:
; сохранить счетчик ВНЕШНЕГО цикла в стеке
        push      cx
; цикл по столбцам (внутренний)----------
        xor       SI,SI     ; смещение <SI>=0 (по столбцам)
        mov       cx,n
        jcxz      @@4
@@2:
        mov       ax,a
        CWD
        add       s0,ax     ; длинное сложение
        adc       s2,dx
; переход к след. элементу массива A в строке
        inc       SI
        inc       SI        ; +2(int!!!)
        LOOP      @@2       ;-------------
; изменение смещения <BX>=<BX>+2*N —
```

```
; переход на следующую строку
        add   bx,N
; восстановить счетчик ВНЕШНЕГО цикла из стека
        pop   cx
        LOOP  @@1 ;=======================================
@@4:
        ret
SummA2  EndP
        End
```

Файл листинга **SummA2.lst** можно посмотреть в прилагаемых к книге материалах.

9.5. Вызов Pascal-процедуры из модуля на Ассемблере

Воспользуемся соглашениями по стыковке модулей **Pascal+Assembler** (см. п. 9.1.2.3) и решим более сложную задачу, но с одномерным массивом.

ПРИМЕР 9.9.

Проанализировать элементы 16-разрядного целочисленного знакового массива **Buffer** длиной не более 100 элементов на предмет их попадания в заданный диапазон **[Min...Max]**. Реализовать вычисления в среде **Borland Pascal+TASM**. Причем, анализ элементов массива сделать в Ассемблере и, если рассматриваемый элемент НЕ попадает в указанный диапазон, сразу же выдать в Паскале сообщение об этом.

Таким образом, делаем главную программу на Паскале — **Check.pas**. Из нее будет происходить вызов подпрограммы на Ассемблере **Check**, из которой, в свою очередь, при необходимости будет вызвана подпрограмма на Паскале **RangeError** с возвратом в точку вызова, т.е. в Ассемблер. Такой задачи мы с вами пока не решали. Но, уверяю вас, все составляющие для ее решения вам уже известны!

Исходный текст программы Check.pas

```
program numbers;
{ CopyRight by Голубь Н.Г. 1993-1998
!!!!!!! Корректность ввода значений переменных и
 элементов массива НЕ проверяется !!!!!
}
{$I check}
{$F+}
var
   Buffer : array [1..100] of integer;
   j,N,Min,Max : integer;
   {!!!! Эта процедура вызывается из АССЕМБЛЕРА !!!!!!!}
    Procedure RangeError(i:integer);
     Begin
        Writeln ('Элемент ', i, ' вышел за границы диапазона [',Min,'..',Max,']!!!');
       Readln;
     End;
   Procedure Check; external;{TASM}
Begin
  Writeln ('Проверка попадания элементов целочисленного массива в диапазон');
  Writeln ('Введите диапазон: Min, Max ');
  Read (Min,Max);
  if Max < Min then
```

```
      begin
        j:=Min;
        Min:=Max;
        Max:=j;
      end;
  N:=0;
  While (NOT Eof AND (N<100)) do
  {выход при ctrl/z или N=100}
  Begin
      N:=N+1;
     Writeln ('Вводите элемент массива [N=',N, '] — окончание ввода ctrl/z или N=100');
      Read ( Buffer [N] );
   End;
   if N<100 then N:=N-1;
   Writeln ('Введены ',N,' элемента(ов) массива:');
      for j:=1 to N do
         begin
        write(Buffer[j],' ');
         if j mod 10 =0 then Writeln;
      end;
    Writeln;
   Writeln ('Начинаем проверку элементов массива в Ассемблере...');
   CHECK ;
   Writeln('Проверка окончена. Остальные элементы — ok! Нажмите любую клавишу.........');
 Readln
End.
```

Исходный текст модуля CHECK.asm

```
           TITLE      CHECK
; CopyRight by Голубь Н.Г. 1993-1998
           .MODEL     LARGE
           LOCALS     @@
           .DATA
           EXTRN      Buffer:Word,    N:Word,    Min:Word,  Max:Word
           .CODE
           EXTRN      RangeError: Far ; ВНЕШНЯЯ процедура
           PUBLIC     CHECK
Arr        EQU        Buffer[BX]       ;      синонимы
Check Proc FAR
           mov        ax,Min
           mov        dx,Max
           xor        bx,bx            ;       смещение <BX>=0
           mov        cx,N
           jcxz       @@4
@@1:       cmp        Arr,ax           ; Buffer[i] < Min ?
; да — выход за ЛЕВУЮ границу диапазона
           jl         @@2
           cmp        Arr,dx           ; Buffer[i] <= Max ?
           jle @@3                     ; да — НОРМА!
@@2:       ; выход элемента Buffer[i] за диапазон
; сохранение регистров перед вызовом внешней процедуры
           push ax bx cx dx
; передача параметра Buffer[i] внешней процедуре
           push Arr
; вызов внешней процедуры
           call RangeError
```

```
; восстановление регистров после отработки внешней
;                    процедуры
        pop      dx cx bx ax
@@3:
; переход к след. элементу массива
        inc      bx
        inc      bx           ; изменение смещения <BX>=<BX>+2
        LOOP     @@1
@@4:
        ret
Check EndP
        End
```

Файл листинга **CHECK.lst** можно посмотреть в прилагаемых к книге материалах.

9.6. Вызов Срр-функции из модуля на Ассемблере

ПРИМЕР 9.10.

Реализуем пример 9.9, но в среде **Borland C++** и **TASM**. Немного усложним его.

1. Пусть наш массив будет длиной максимум в 1000 элементов. Поэтому, как и в примере 9.7, применим генерацию элементов массива через датчик случайных чисел, если число элементов массива будет превышать 9.
2. Реализацию кода в Ассемблере немного оптимизируем за счет использования более быстрой *регистровой адресации* вместо *адресации по базе BX со смещением.* Если таких режимов адресации в программе используется много, мы можем немного уменьшить длину машинного кода — см. соответствующие LST-файлы.
3. Опишем вызываемую из Ассемблера процедуру с параметрами:

void RangeError(int i, int buf);

Она должна быть оттранслирована компилятором в стиле С (почему? — см. п. 6.1.2.1).

4. Вызов из Ассемблера этой процедуры сделаем в двух вариантах:

- **вариант 1** — явная передача в стек параметров и очистка стека после вызова процедуры **RangeError**;
- **вариант 2** — использование команды **CALL** с расширенным синтаксисом (см. п. 9.1.2.1).

Исходный текст программы Check2.cpp (вар.1 и вар.2)

```
/* Check2.cpp Borland C++ 5.02
 Проект !!!!!! DOS (standard) !!!!!! asm+cpp файлы
        (c) Copyright 1998-2000, 2001 by Голубь Н.Г.
Генерация значений элементов массива через датчик случайных чисел, если n>=10.
Проверка введенных данных на допустимый диапазон.
Проверить в Ассемблере попадание элементов целочисленного массива в диапазон.
Вызов функции вывода сообщения об ошибке из Ассемблера
*/

#include <iomanip.h> // манипулятор setw(16)
#include <stdlib.h> // int random(int num);
#include <time.h> // void randomize(void);
#include <conio.h> //getch()
```

```
#include "convert.h"
const int N=1000;
const char* CHECK_ARRAY
   = TXT("\nПроверка попадания элементов целочисленного массива в диапазон\n");
const char* CHECK_ASM
   = TXT("\nНачинаем проверку элементов массива в Ассемблере...\n");
const char* EXIT_OK = TXT("\nПроверка окончена. Остальные элементы — ok!\n");
const char* RANGE = TXT("\nВведите диапазон: Min, Max \n");
const char* ELEMENT = TXT("\nЭлемент [");
const char* MSG = TXT(" вышел за границы диапазона [");
int Buffer[N];
int n, Min, Max;
extern "C"
   {
   void CheckCA (void);
   void RangeError(int i, int buf);
   }
void RangeError(int i, int buf)
{
   cout << ELEMENT << i <<"]" << "=" << buf << MSG << Min << ".."
        << Max << "]!!!\n";
   getch();
}
void inputN(const int N, int& n)
{ do{ cout << "\n??? n [1.."<< N <<"]=======>";
        cin >> n;
        cout << "inputN: n = " << n << endl;
   } while (n<1 || n>N);
}

void inputA(int A[], int n)
{ randomize();
  int j=1;
  long temp;
  for (int i=0; i<n; i++)
        if (n<10)
          { cout << "??? A["<< i <<"]=======>";
            cin >> temp;
            A[i] = (int)temp;
          if (A[i]!=temp)i--; // !!!!!!! проверка на диапазон !!!!!!!!!!!!!!
          }
        else
          { j = -j;
           A[i]= j*(int)random (300);
          }
}

void outputA(const int A[], int n)
{ cout << "\n=================================================================\n"
        << " Array: \n";
  for (int i=0; i<n; i++)
          cout << setw(8) << A[i];
  cout << endl;
}
```

```
void main()
{ int t=1;
   for (;;)
     { cout << "\n=============== Test #" << t++ << " ===============";
       cout << CHECK_ARRAY;
       cout << RANGE; cin >> Min >> Max;
       if (Max < Min)
          { int j=Min;
            Min=Max;
            Max=j;
          }
 // ввод реального количества элементов массива A
 inputN(N,n);
 // ввод элементов массива A
  inputA(Buffer,n);
  outputA(Buffer,n);
 cout << CHECK_ASM;
 CheckCA();
 cout << EXIT_OK;
 cout << "\n Exit Ctrl-C";
     }
}
```

Исходный текст модуля checkc.asm — вариант 1

```
      TITLE      CHECK
; CopyRight by Голубь Н.Г. 1993-1998, 2001
; ЯВНАЯ передача параметров ВНЕШНЕЙ процедуре
      .MODEL  LARGE,C
      LOCALS  @@
      .DATA
      EXTRN      C Buffer:Word, n:Word
      EXTRN      C Min:Word, Max:Word
      .CODE
      EXTRN      C RangeError: Far ; ВНЕШНЯЯ процедура
      PUBLIC     C CheckCA
k     dw         ?          ; счетчик элементов массива
Arr EQU Buffer[BX]          ; синонимы
CheckCA Proc far
      mov        ax,Min
      mov        dx,Max
      mov        k,0
      xor        bx,bx ; смещение <BX>=0
      mov        cx,n
      jcxz       @@4
@@1:
      mov        DI,Arr
      cmp        DI,ax      ; Buffer[k] < Min ?
      jl         @@2        ; да — выход за ЛЕВУЮ границу диапазона
      cmp        DI,dx      ; Buffer[k] <= Max ?
      jle        @@3        ; да — НОРМА!
@@2:                        ; выход элемента Buffer[k] за диапазон
; сохранение регистров перед вызовом внешней процедуры
      push       ax bx cx dx
```

```
; передача параметров внешней процедуре
; (в ОБРАТНОМ порядке!!!!!!!!)
; void RangeError(int i, int buf);
     push       DI    ; элемент массива
     push       k     ; его индекс
; вызов внешней процедуры
     call       RangeError
     add        sp,4  ;ОЧИСТКА СТЕКА (4 байта!!!)
; восстановление регистров
; после отработки внешней процедуры
     pop        dx cx bx ax
@@3:
; переход к след. элементу массива
     inc        bx
     inc        bx    ; изменение смещения <BX>=<BX>+2
     inc        k     ; k++
     LOOP       @@1
@@4:
     ret
CheckCA EndP
     End
```

В варианте 2 (с применением расширенного синтаксиса команды **CALL**) на Ассемблере изменится только вызов функции.

Фрагмент исходного текста модуля checkc.asm — вариант 2

```
; сохранение регистров перед вызовом внешней процедуры
         push       ax bx cx dx
; void RangeError(int i, int buf);
; РАСШИРЕННЫЙ синтаксис команды CALL:
; передача параметров внешней процедуре
; (в НОРМАЛЬНОМ порядке!!!!!!!!)
         call       RangeError C,k,DI
; восстановление регистров
; после отработки внешней процедуры
         pop        dx cx bx ax
```

Посмотрим теперь на **LST**-файлы. Сравним сначала реализации вызова внешней функции **RangeError** в первом и втором вариантах.

Фрагмент файла checkc.lst — вариант 1

```
                      ; ЯВНАЯ передача параметров ВНЕШНЕЙ процедуре
0000                            .MODEL LARGE,C
                                LOCALS @@
0000                            DATA
                        EXTRN C Buffer:Word, n:Word, Min:Word, Max:Word
0000                            .CODE
                                EXTRN C RangeError: Far ; ВНЕШНЯЯ процедура
                                PUBLIC C CheckCA
0000 ????           k        dw    ?      ; счетчик элементов массива
 =       Buffer[BX]  Arr EQU Buffer[BX]   ; синонимы
0002                 CheckCA Proc far
0002 A1 0000e                mov   ax,Min
0005 8B 16 0000e             mov   dx,Max
```

```
   14  0009 2E: C7 06 0000r 0000   mov   k,0
   15  0010 33 DB                  xor   bx,bx ; смещение <BX>=0
   16  0012 8B 0E 0000e            mov   cx,n
   17  0016 E3 2A                  jcxz  @@4
   18  0018                   @@1:
   19  0018 8B BF 0000e            mov   DI,Arr
   20  001C 3B F8                  cmp   DI,ax   ; Buffer[k]     < Min ?
   21  001E 7C 04              jl  @@2 ; да — выход за ЛЕВУЮ границу диапазона
   22  0020 3B FA                  cmp   DI,dx   ; Buffer[k]<= Max ?
   23  0022 7E 15                  jle @@3       ; да — НОРМА!
   24  0024                   @@2: ; выход элемента Buffer[k] за диапазон
   25                 ; сохранение регистров перед вызовом внешней процедуры
   26  0024 50 53 51 52            push ax      bx cx dx
   27   ; передача  параметров внешней процедуре (в   ОБРАТНОМ порядке!!!!)
   28                         ; void RangeError(int i, int buf);
   29  0028 57                     push  DI      ; элемент массива
   30  0029 2E: FF 36 0000r        push  k       ; его индекс
   31  002E 0E E8 0000e            call  RangeError ; вызов внешней процедуры
   32  0032 83 C4 04               add   sp,4    ;ОЧИСТКА СТЕКА (4 байта!!!)
   33   ; восстановление регистров после отработки внешней  процедуры
   34  0035 5A 59 5B 58            pop   dx      cx bx ax
   35  0039                   @@3:
   36                         ; переход к след. элементу массива
   37  0039 43                     inc   bx
   38  003A 43                     inc   bx      ; изменение смещения <BX>=<BX>+2
   39  003B 2E: FF 06        0000r inc   k       ; k++
   40  0040 E2 D6                  LOOP @@1
   41  0042                   @@4:
 1 42  0042 CB                     RET 00000h
   43  0043                   CheckCA EndP
   44                                 End
```

Фрагмент файла checkc.lst — вариант 2

```
   25                         ; сохранение регистров перед вызовом внешней процедуры
   26  0024 50 53 51 52            push ax       bx cx dx
   27                         ; РАСШИРЕННЫЙ синтаксис команды CALL
   28                         ; void RangeError(int i, int buf);
   29                 call RangeError    C,k,DI  ; вызов внешней процедуры
 1 30  0028 57                     PUSH DI
 1 31  0029 2E: FF 36        0000r PUSH k
 1 32  002E 0E E8 0000e            CALL  RangeError
 1 33  0032 83 C4 04               ADD SP,00004h
   34                         ; передача   параметров внешней процедуре
   35                         ;      (в НОРМАЛЬНОМ порядке!!!!!!!!)
   36                         ; восстановление регистров после отработки внешней процедуры
   37  0035 5A 59 5B 58            pop dx        cx bx ax
```

Из последнего листинга совершенно четко видно, что команда **CALL** с расширенным синтаксисом — это не что иное, как **макрокоманда**, которая порождает точно такое же ***макрорасширение***, как и при явной передаче параметров. Но, несомненно, человеку такая команда более понятна. Тем более, что она учитывает все "языковые" нюансы...

Теперь посмотрим, что дала замена *адресации по базе BX со смещением* на *регистровую адресацию*. В этом плане машинный код для варианта 1 и варианта 2 разный (файл **checkc.lst**).

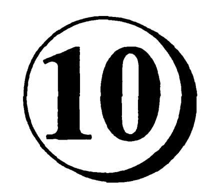
Фрагмент файла checkc.lst — вариант 1 (пример 9.10).

```
10    =    Buffer[BX]  Arr EQU Buffer[BX]  ; синонимы
.................................................................................
18   0018                  @@1:
19   0018 8B BF 0000e          mov DI,Arr
20   001C 3B F8                cmp DI,ax ; Buffer[k]    < Min ?
21   001E 7C 04            jl  @@2 ; да — выход за ЛЕВУЮ границу диапазона
22   0020 3B FA                cmp DI,dx ; Buffer[k]    <= Max ?
23   0022 7E 15                jle @@3 ; да — НОРМА!
```

Фрагмент файла check.lst (пример 9.9).

```
8     =    Buffer[BX]  Arr EQU Buffer[BX]  ; синонимы
.................................................................................
15   000F 39 87 0000e      @@1: cmp Arr,ax ; Buffer[i] < Min ?
16   0013 7C 06            jl  @@2 ; да — выход за ЛЕВУЮ границу диапазона
17   0015 39 97 0000e          cmp Arr,dx ; Buffer[i] <= Max ?
18   0019 7E 10                jle @@3 ; да — НОРМА!
19   001B                  @@2: ; выход элемента Buffer[i]   за диапазон
```

Если хотите, можете потренироваться в расшифровке соответствующих машинных кодов, чтобы лучше понять логику работы компьютера.



Команды управления состоянием микропроцессора i8086

Если сам прям, то все исполнят и без приказания. А если сам не прям, то слушаться не будут, даже если им и прикажут.

Конфуций (ок. 551–479 гг. до н.э.)

Все эти команды НЕ имеют операндов, поскольку они им НЕ нужны, и занимают в машинном коде один байт. Действия команд понятны из их мнемонического кода (человеку) и из машинного кода операции (компьютеру).

10.1. Команды управления флагами

С некоторыми командами управления флагами мы с вами уже встречались — см. п. 5.1.4.3 и п. 5.1.5. В этом параграфе мы познакомимся с остальными командами этой группы.

Таблица 10.1. Команды установки (SeT) или сброса (CLear) флагов DF, IF, CF.

Команда	Действие команды (состояние флагов после выполнения команды)		
	DF	IF	CF
STC	-	-	1
CLC	-	-	0
STD	1	-	-
CLD	0	-	-
STI	-	1	-
CLI	-	0	-

С командами установки и сброса флага направления (**DF**) мы более подробно познакомимся в следующей, 11 главе, поскольку именно этот флаг отвечает за направление обработки строк.

Команда **STI** устанавливает флаг разрешения прерывания (**IF**) в единицу, разрешая процессору распознавать *маскированные прерывания*. Другие флаги не меняются. *Немаскированные прерывания* распознаются процессором всегда, независимо от значения флага **IF**.

Есть еще две команды работы с флагом переноса **CF**:

CMC (CoMplement Carry flag — инвертирование флага переноса) и **SALC** (Set register AL according to CF — установить регистр AL в соответствии с флагом CF).

Команда **SALC** относится к числу недокументированных команд. Она имеет машинный код **0D6h**.

Логика работы команды:
If <CF>=1 then AL=0FFh else AL=00h

Это эквивалентно команде (но БЕЗ изменения флагов):
SBB AL,AL

10.2. Команды внешней синхронизации

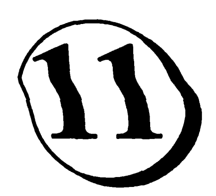
Команды этой группы НЕ влияют на флаги.

HLT (HaLT — останов). Эта команда производит останов ЦП и переводит его в состояние ожидания сигнала сброса (RESET) или сигнала *немаскированного прерывания* (INTR).

Команда **WAIT** (пауза) переводит процессор в состояние ожидания, пока не поступит сигнал от сопроцессора об окончании обработки последней команды. Таким образом, команда **WAIT** предназначена для синхронизации работы процессора и более медленного сопроцессора. Начиная с сопроцессора **i387**, такая синхронизация выполняется автоматически.

Команда **NOP** (No OPeration — нет операции) является пустым оператором. Она часто используется в целях подгонки времени при моделировании таймера (программная задержка), для выравнивания памяти и как «держатель места». С ней в последнем качестве мы с вами уже встречались — см. п. 9.1.1 (пример 9.1 и его анализ).

LOCK (LOCK signal prefix — префикс выдачи сигнала LOCK). **LOCK** — это однобайтный префикс, который может предшествовать любой команде. **LOCK** заставляет процессор выработать сигнал блокировки системной шины на время выполнения последующей команды. Использование сигнала блокировки делает шину недоступной для любого внешнего устройства или события, включая прерывания и передачу данных. Эта команда была предусмотрена для поддержки мультипроцессорных систем с разделенными ресурсами. Ее наиболее часто используют перед командами **XCHG**, **MOV** и **MOVS** с целью предотвращения прерывания операций по пересылке данных. Например, **LOCK XCHG BL,AL**



Основные команды обработки строк для IBM PC XT/AT

Суть дела не в полноте знания, а в полноте разумения.

Демокрит (около 470 или 460 гг. до н.э. — умер в глубокой старости)

До сих пор мы работали в основном с целочисленными данными. Команды, показанные в предыдущих главах, оперировали одним байтом, или одним словом за одно выполнение. Часто, однако, бывает необходимо переслать или сравнить поля данных, которые превышают по длине одно слово. Это можно сделать в Ассемблере по-разному, в зависимости от того, какой смысл мы вкладываем в такие поля данных. В этой главе нас будут интересовать команды, которые позволяют обрабатывать **цепочку данных** как одно целое. Элементы такого формата обычно известны как строковые данные и могут являться в Ассемблере как символьными или числовыми, так и любыми другими. Понимание принципов работы этих команд и умение грамотно их использовать может очень облегчить написание программ на Ассемблере и даже сделать их более эффективными.

Из алгоритмических языков, надеюсь, вам известны такие структуры, как строки. В каждом алгоритмическом языке они имеют свои особенности и свой внутренний формат. С внутренним форматом строк, принятым в алгоритмическом языке Паскаль, мы и познакомимся в этой главе.

Что касается языка C/C++, он имеет два способа представления и обработки строк. Первый способ позаимствован из языка C (строки в стиле C) и другой, альтернативный способ, основанный на библиотеке класса *string* (в реализации Borland C++ есть еще устаревшая библиотека для этого класса — **cstring.h**) — строки в стиле C++. Строки в стиле C (*C-строки*) — это *массивы символов (каждый символ длиной один байт), ограниченные символом* **нуль "\0"**. Это обстоятельство имеет *исторические корни*. Известно, что язык C был создан сотрудником фирмы Bell Labs Деннисом Ритчи в 1972 г. во время его совместной работы с Кеном Томпсоном над операционной системой Unix. Операционная си-

стема Unix создавалась на базе компьютеров PDP-11 американской фирмы DEC (Digital Equipment Corporation). В системе команд этих компьютеров были команды обработки строк типа ASCIIZ (т.е. строк, состоящих из символов ASCII, оканчивающихся символом ZERO — нуль). Таким образом, фактическая длина С-строки всегда на единицу больше.

Считается, что **С-строки в высшей степени эффективны**, но пользоваться ими не всегда удобно. Поэтому программисты и различные среды программирования (компиляторы) на протяжении ряда лет разрабатывали свои версии обработки строк, которые, естественно, отличались друг от друга. В конечном итоге, все эти дебаты увенчались разработкой в 1998 г. стандарта ISO/IEC 14882 "Standard for the C++ Programming Language", и появились более удобные и естественные операции для работы со строками, использующие концепцию объектно-ориентированного программирования и заложенные в стандартную библиотеку шаблонов **STL**. Для эффективной работы с С-строками программист должен хорошо понимать, что они собой представляют, и владеть стандартными функциями обработки строк, которые сосредоточены в основном в библиотеке **string.h**. А это — опять знание основ обработки строк в Ассемблере.

11.1. Общие положения по обработке строк в Ассемблере

1. В мнемонике команд обработки строк всегда содержится буква **S** (String — строка). Она является последней или предпоследней буквой.
2. Содержимое строки для микропроцессора НЕ имеет никакого значения. Это могут быть символы, числа и все, что угодно. Основное, что имеет значение, — это длина операнда.
3. Строка в **базовом Ассемблере** может обрабатываться побайтно (**BYTE** — последняя буква в команде будет **B**) или пословно (**WORD** — последняя буква в команде будет **W**).
4. Строка может обрабатываться группой (цепочкой), тогда перед командой появляется префикс **REPx** (REPeat — повторить). Количество повторений должно находиться в регистре **CX**. Этот префикс алгоритмически подобен команде **LOOPx** (см. п. 9.3).
5. *Строка-приемник* должна находиться **обязательно** в дополнительном сегменте памяти **ES** со смещением **DI** (адресация <**ES:DI**>).
6. *Строка-источник* должна находиться в сегменте данных **DS** со смещением **SI** (адресация <**DS:SI**>). Допускается замена регистра сегмента **DS** с помощью префикса замены сегмента.
7. В процессе *циклического* выполнения команд *указатели* **SI** и **DI** автоматически модифицируются в зависимости от длины элемента строки и значения флага направления **DF**:

   Если <**DF**>=**0**, значения **SI** и **DI** увеличиваются (**строка обрабатывается слева направо** — в сторону **больших** адресов).

   Если <**DF**>=**1**, значения **SI** и **DI** уменьшаются (**строка обрабатывается справа налево** — в сторону **меньших** адресов).
8. Флаг направления **DF** очищается или устанавливается, соответственно, командами **CLD** или **STD** (см. п. 10.1).
9. Длина строки в **базовом Ассемблере** <=64K байт.

10. Команды сравнения или сканирования строк устанавливают значения флагов аналогично команде **CMP** (см. табл. 5.8).

Таблица 11.1. Строковые операции (цепочечные команды).

Команда	Назначение	Алгоритм работы
Команды пересылки MOVSx		
MOVS *приемник, источник*	Копирование строки	*Приемник* <== *источник*
MOVSB	Копирование строки байтов	[DI] <== [SI]
MOVSW	Копирование строки слов	
Команды сравнения CMPSx		
CMPS *приемник, источник*	Сравнение строк	*Приемник* ~ *источник*
CMPSB	Сравнение строк байтов	[DI] ~ [SI]
CMPSW	Сравнение строк слов	
Команды сканирования SCASx		
SCAS *приемник*	Сканирование строки	*Приемник* ~ **AX** (или **AL**)
SCASB	Сканирование строки байтов	[DI] ~ AL
SCASW	Сканирование строки слов	[DI] ~ AX
Команды загрузки LODSx		
LODS *источник*	Чтение из строки	*Источник* ===> **AX** (**AL**)
LODSB	Чтение байта из строки	[SI] ===> AL
LODSW	Чтение слова из строки	[SI] ===> AX
Команды сохранения STOSx		
STOS *приемник*	Запись в строку	**AX** (или **AL**) ==> *приемник*
STOSB	Запись байта в строку	AL ===> [DI]
STOSW	Запись слова в строку	AX ===> [DI]
Префиксы повторения REPx		
REP	Повторять команду	
REPE/REPZ	Повторять команду, пока равно (флаг ZF=1)	
REPNE/REPNZ	Повторять команду, пока НЕ равно (флаг ZF=0)	

11.2. Особенности обработки Ассемблером строк Borland Pascal

1. Строка (**string**) состоит из символов длиной один байт.
2. Длина строки <255 байт.
3. В первом байте (невидимом для программиста на Паскале) всегда размещается значение фактической длины строки.
4. Если строки описаны как глобальные и передаются в Ассемблер, они в соответствии с соглашениями (см. п. 6.1.1.2) находятся в сегменте данных. Поэтому для обработки таких строк в Ассемблере нужно установить значение сегментных регистров **ES=DS**:

```
PUSH DS
POP ES
```

5. Далее следует установить значение флага направления (команды **CLD** или **STD**) и загрузить адреса обрабатываемых строк:

```
LEA SI, StrSource ; адрес строки-источника
LEA DI, StrDestination ; адрес строки-приемника
```

6. По окончании обработки строк, если это нужно, в первый байт результирующей строки нужно занести ее фактическую длину, например, с помощью следующей команды:

```
MOV [StrDestination], AL
```

ПРИМЕР 11.1.

Промоделировать на Ассемблере основные процедуры и функции обработки Паскаль-строк:

- **Delete(var S:String; Poz,M:Integer);** — удаление из строки **S M** символов, начиная с номера позиции **Poz**.
- **Insert(Source:String; Var S:String; Poz:Integer);** — вставка подстроки **Source** в строку S, начиная с номера позиции **Poz**.
- **Copy(S:String; Poz,M:Integer):String;** — копирование из строки **S M** символов, начиная с номера позиции **Poz**.
- **Pos(S1, S2:String):Byte;** — поиск номера позиции, начиная с которого подстрока **S1** входит в строку **S2**.

Особые ситуации, связанные с некорректностью ввода номера позиции **Poz** или количества символов **M**, будем обрабатывать в Паскале. Попробуйте в этом примере разобраться. В нем применены практически все перечисленные в табл. 11.1 команды.

Исходный текст программы String.pas

```
Program stringP;
{---------------------------------------------------------------
   Лекционный пример 11.1.
  Моделирование на Ассемблере функций обработки строк
  Корректность ввода символов НЕ проверяется.
  CopyRight by Голубь Н.Г., 1993, 2001
---------------------------------------------------------------}
{$L strdel }
{$L strcop }
{$L strins }
{$L strpos }
Label 1,2,3;
Var
  StrS,StrD,StrIns: string;
  N,M,Poz,L : integer;
Procedure del; FAR; External;
Procedure inst; FAR; External;
Procedure cop; FAR; External;
Procedure posstr; FAR; External;
Begin
  Writeln ('===== Моделирование на Ассемблере функций обработки строк =====');
  Writeln('Введите исходную строку');
  ReadLn(StrS);
  N:= Length(StrS);
```

```
        { ****** УДАЛЕНИЕ *********** }
 Writeln('Сколько символов удалять?');
 ReadLn(M);
 1: Writeln('С какой позиции?');
 ReadLn(Poz);
 if Poz > N then
  begin
        Writeln('Данное НЕ корректно, ПОВТОРИТЕ ввод!');
        goto 1
  end;
 if M > N — Poz + 1 then
  begin
        Writeln('Данное НЕ корректно, будут удалены символы до конца строки');
        M := N — Poz + 1
  end;
 DEL;
 Writeln('Результат');
 Writeln('Assembler:',StrD);
 StrD:= StrS;
 DELETE (StrD,Poz,M);
 Writeln('Pascal: ',StrD);
 ReadLn;
 StrD:='';

        { ****** ВСТАВКА *********** }
 Writeln('Введите вставляемую строку');
 ReadLn(StrIns);
 L:= Length(StrIns);
 2: Writeln('С какой позиции вставлять?');
 ReadLn(Poz);
 if Poz > N then
  begin
        Writeln('Данное НЕ корректно, ПОВТОРИТЕ ввод!');
        goto 2
  end;
 if N + L > 255 then
  begin
        Writeln('Результирующая строка усекается до 255 символов');
        L:= 255 — N
  end;
 INST;
 Writeln('Результат');
 Writeln('Assembler:',StrD);
 StrD:= StrS;
 INSERT (StrIns,StrD,Poz);
 Writeln('Pascal: ',StrD);
 ReadLn;
 StrD:='';

        { ****** КОПИРОВАНИЕ *********** }
 Writeln('Сколько символов копировать?');
 ReadLn(M);
```

```
 3: Writeln('С какой позиции?');
 ReadLn(Poz);
 if Poz > N then
  begin
        Writeln('Данное НЕ корректно, ПОВТОРИТЕ ввод!');
        goto 3
  end;
 if M > N — Poz + 1 then
  begin
        Writeln('Данное НЕ корректно, будут скопированы символы до конца строки');
        M := N — Poz + 1
  end;
 COP;
 Writeln('Результат');
 Writeln('Assembler:',StrD);
 StrD:= Copy(StrS,Poz,M);
 Writeln('Pascal: ',StrD);
 ReadLn;
 StrD:='';
        { ****** ПОЗИЦИЯ СОВПАДАЮЩИХ СИМВОЛОВ *********** }
 Writeln ('Вычисление позиции, начиная с которой эталонная строка совпадает с исходной');
 Writeln('Введите эталонную строку');
 ReadLn(StrIns);
 L:= Length(StrIns);
 POSSTR;
 Writeln('Результат');
 Writeln('Assembler:',Poz);
 Poz := Pos(StrIns,StrS);
 Writeln('Pascal: ',Poz);
 ReadLn
END.
```

Исходный текст модуля StrDel.asm

```
        TITLE DELETE STRING
; CopyRight by Голубь Н.Г., 1993, 2001
; модель аналогичной процедуры TPascal:
; DELETE (StrD,Poz,M);
; Корректность данных M (кол-во удаляемых символов)
;          и Poz НЕ проверяется!!!!!!!
        .MODEL LARGE
        .DATA
   EXTRN StrS:Byte, StrD:Byte
   EXTRN N:Word, M:Word, Poz:Word
   .CODE
        PUBLIC     DEL
Del     Proc       far
        Push       DS
        Pop        ES               ; ES=DS
        CLD           ; DF=0 (обработка символов —> )
        Lea        SI,StrS          ; адрес исх. строки
        Lea        DI,StrD          ; адрес рез. строки
        Mov        CX,Poz
; пересылка Poz символов, изменение SI,DI
Rep     MovSB
        Mov             CX,M
; пропуск M символов в исх.строке, изменение SI
```

```
Rep     LodSB
        Mov   CX,N
        Sub   CX,M
        Sub   CX,Poz
        Inc   CX           ; остаток = N - M - (Poz - 1)
; пересылка оставшихся символов исх. строки
Rep     MovSB
        Mov   CX,M
        Mov   Al,' '
Rep     StoSB              ; очистка «остатка»
        ret
Del     EndP
        End
```

Исходный текст модуля StrIns.asm

```
      TITLE INSERT STRING
;CopyRight by Голубь Н.Г., 1993, 2001
; модель аналогичной процедуры TPascal:
;     INSERT (StrIns,StrD,Poz);
; Корректность данных и Poz НЕ проверяется!!!!!!!!
          .MODEL LARGE
          .DATA
      EXTRN StrS:Byte, StrD:Byte, StrIns:Byte
      EXTRN N:Word, Poz:Word,L:Word
          .CODE
          PUBLIC  INST
INST      Proc    far
          Push    DS
          Pop     ES        ; ES=DS
          CLD               ; DF=0 (обработка символов --> )
          Lea     SI,StrS   ; адрес исх. строки
          Lea     DI,StrD   ; адрес рез. строки
          Mov     CX,Poz
Rep       MovSB             ; пересылка Poz символов, изм. SI,DI
          Push SI           ; сохранение указателя StrS

          Mov     CX,L
          Lea     SI,StrIns; адрес ВСТАВЛЯЕМОЙ строки
          Inc SI            ; адрес информационной части
Rep       MovSB             ; пересылка L символов, изм. SI,DI
          Pop     SI        ; восстановление указателя StrS
          Mov     CX,N
          Sub     CX,Poz
          Inc     CX        ; остаток = N - (Poz - 1)
; пересылка оставшихся символов исх. строки
Rep       MovSB
          Mov     AX,N
          Add     AX,L
; фактическая длина строки = N + L
          Mov     [StrD],AL
          ret
INST      EndP
          End
```

Исходный текст модуля StrCop.asm

```
 TITLE COPY STRING
;CopyRight by Голубь Н.Г., 1993, 2001
; модель аналогичной функции TPascal:
;       StrD:=Copy(StrS,Poz,M);
; Корректность данных M (кол-во копируемых символов)
;            и Poz НЕ проверяется!!!!!!!!
        .MODEL LARGE
        .DATA
        EXTRN StrS:Byte, StrD:Byte, M:Word, Poz:Word
        .CODE
        PUBLIC COP
cop     Proc       far
        Push       DS
        Pop        ES          ; ES=DS
        CLD   ; DF=0 (обработка символов —> )
        Lea        SI,StrS     ; адрес исх. строки
        Lea        DI,StrD     ; адрес рез. строки
        Mov        CX,Poz
Rep     LodSB                  ; установка на символ Poz, изм. SI
        Inc        DI          ; установка на инф. часть
        Mov        CX,M
Rep     MovSB                  ; копирование M символов, изм. SI,DI
        Mov        AX,M
; длина выходной строки в первый байт
        Mov        [StrD],AL
        ret
Cop     EndP
        End
```

Исходный текст модуля StrPos.asm

```
 TITLE POSITION STRING
; CopyRight by Голубь Н.Г., 1993, 2001
; модель аналогичной процедуры TPascal:
;       Poz:=Pos (StrIns,StrS);
; Корректность данных и Poz НЕ проверяется!!!!!!!!
        .MODEL     LARGE
        .DATA
        EXTRN      StrS:Byte, StrIns:Byte, N:Word
        EXTRN      Poz:Word,L:Word
        .CODE
        PUBLIC     POSSTR
POSSTR  Proc       far
        Mov        Poz,0       ; строка НЕ найдена
        Push       DS
        Pop        ES          ; ES=DS
        CLD                    ; DF=0 (обработка символов —> )
        Lea        DI,StrS     ; адрес исх. строки
        Lea        SI,StrIns   ; адрес эталонной строки
        Inc        SI
        Inc        DI          ; инф. часть
        Mov        DX,SI
        Mov        BX,1        ; счетчик позиций исх. строки
```

```
        Mov     BP,N
        Sub     BP,L
; N - L + 1 - ограничитель для исх. строки
        Inc     BP
next:
; восстановление нач. адреса эталона
        Mov     SI,DX
        Mov     CX,L
; запоминание тек. адреса исх. строки
        Push    DI
RepE    CmpSB       ; проверяем элементы на совпадение
; восстановление тек. адреса исх. строки
        Pop     DI
        Je      exit
        Inc     DI          ; переход к след. символу
        Inc     BX
        Cmp     BX,BP
        JLE     next
        ret
; найдены совпадающие элементы
exit:   Mov     Poz,BX
        ret
POSSTR  EndP
        End
```

Результат работы данной программы:

```
===== Моделирование на Ассемблере функций обработки строк =====
Введите исходную строку
1234567890asdfghjkl1111111111111111222222222222
Сколько символов удалять?
150
С какой позиции?
30
Данное НЕ корректно, будут удалены символы до конца строки
Результат

Assembler:1234567890asdfghjkl1111111111
Pascal: 1234567890asdfghjkl1111111111

Введите вставляемую строку
WEEKEND
С какой позиции вставлять?
200
Данное НЕ корректно, ПОВТОРИТЕ ввод!
С какой позиции вставлять?
20
Результат

Assembler:1234567890asdfghjklWEEKEND1111111111111111222222222222
Pascal: 1234567890asdfghjklWEEKEND1111111111111111222222222222

Сколько символов копировать?
120
С какой позиции?
35
Данное НЕ корректно, будут скопированы символы до конца строки
Результат
```

```
Assembler:12222222222222
Pascal: 12222222222222

Вычисление позиции, начиная с которой эталонная строка совпадает с
   исходной
Введите эталонную строку
11111111
Результат

Assembler:20
Pascal: 20

===== Моделирование на Ассемблере функций обработки строк =====
Введите исходную строку
12345qazwsxedc0987654321
Сколько символов удалять?
4
С какой позиции?
10
Результат

Assembler:12345qazwc0987654321
Pascal: 12345qazwc0987654321

Введите вставляемую строку
weekend
С какой позиции вставлять?
12
Результат

Assembler:12345qazwsxweekendedc0987654321
Pascal: 12345qazwsxweekendedc0987654321

Сколько символов копировать?
5
С какой позиции?
1
Результат

Assembler:12345
Pascal: 12345

Вычисление позиции, начиная с которой эталонная строка совпадает с
   исходной
Введите эталонную строку
weekend
Результат

Assembler:0
Pascal: 0
```



Основные особенности процессоров i386, i486, Pentium

Все в мире повторяется, и возвращается ветер на круги своя.

Экклезиаст

До этой главы мы с вами, уважаемые читатели, изучали в основном команды и директивы процессора **i8086** и несколько новых команд процессора **i286**. Мы знаем, что семейство процессоров фирмы **Intel** и совместимых с ними продолжает бурно развиваться, наращивая и память, и быстродействие (см. табл. 3.1). Появляются новые команды, новые идеи по повышению эффективности обработки процессором колоссального объема информации.

С точки зрения *идеологии программирования* процессор **i8086** является базовой моделью, а процессор **i386** является первым процессором, реализующим в полном объеме многозадачность и 32-разрядное программирование. Он использовал много технических новшеств и стал на порядок сложнее, чем процессор **i8086**. Принципиально новым для него является появление 32-разрядных регистров, возможность адресации в пределах 4Г байт оперативной памяти и дополнительных, аппаратно реализованных, режимов работы: режимы виртуальной памяти и многозадачности (защищенный режим).

В процессоре **i486** добавлено несколько новых команд и появляется внутренний сопроцессор. Именованные процессоры **Pentium** и их клоны исподволь готовили и осваивали новые технические решения — см. табл. 3.1.

Следующий *революционный шаг* сделал процессор **Intel Pentium IV**, но нам, начинающим осваивать Ассемблер, пока трудно будет оценить всю меру технической революционности — надо набраться опыта... Но никто не мешает любознательному и уже кое-что понимающему читателю посмотреть и оценить свои силы, познакомившись в Интернете со всеми свежими новостями для разработчиков программного обеспечения на следующих **основных** сайтах:

- официальный сайт компании Intel - документация для разработчиков "Intel® Pentium® 4 Processors – Manuals" (http://developer.intel.com/design/pentium4/manuals/);
- официальный сайт компании AMD (http://www.amd.com:7246/us-en/Processors/DevelopWithAMD/);
- техническая документация компании AMD (http://www.amd.com/us-en/Processors/TechnicalResources/, http://www.amd.com/us-en/Support/);
- русскоязычный сайт компании Intel (http://developer.intel.ru/design/);
- русскоязычный сайт компании AMD (http://www.amd.ru/).

Информации на официальных сайтах достаточно много и без предварительной подготовки (и даже при условии знания английского языка) в ней ОЧЕНЬ легко утонуть... Поэтому будем продолжать учиться плавать.

12.1. Директивы текущего типа (со)процессора

Чтобы компилятор правильно распознавал новые команды и директивы (инструкции), в синтаксисе языка Ассемблера есть директивы для указания типа процессора или сопроцессора. Если ни одну из этих директив НЕ указывать, то компилятор считает, что происходит работа с базовым процессором **i8086** (по концепции умолчания).

Перечислим эти директивы, используя общеупотребительный синтаксис MASM. Соответствие их синтаксису TASM см. в прил. 5.

Таблица 12.1. Директивы указания текущего типа (со)процессора.

Директива	Назначение	Примечание
.8086	Разрешены инструкции базового процессора i8086 (и идентичные им инструкции процессора i8088). Запрещены инструкции более поздних процессоров.	Директива используется по умолчанию.
.186	Разрешены инструкции соответствующего процессора ix86 (x=1,...,6). Запрещены инструкции более поздних процессоров.	
.286		
.386		
.486		
.586		
.686		Поддерживается, начиная с версии 5.3. TASM32.exe

Таблица 12.1. (продолжение)

.8087	Разрешены инструкции соответствующего сопроцессора ix87 наряду с инструкциями процессора. Запрещены инструкции более поздних сопроцессоров и процессоров.	
.287		
.387		
.487		
.587		
.286c	Разрешены НЕПРИВИЛЕГИРОВАННЫЕ инструкции соответствующего процессора ix86 (x=2,...,6) и сопроцессора ix87 (x=2,...,5). Запрещены инструкции более поздних процессоров и сопроцессоров.	
.386c		
.486c		
.586c		
.686c		Поддерживается, начиная с версии 5.3. TASM32.exe
.286p	Разрешены ВСЕ инструкции соответствующего процессора ix86 (x=2,...,6), включая привилегированные команды и инструкции сопроцессора ix87 (x=2,...,5). Запрещены инструкции более поздних процессоров и сопроцессоров.	
.386p		Поддерживается ТОЛЬКО 32-разрядными компиляторами TASM32.exe
.486p		
.586p		
.686p		Поддерживается, начиная с версии 5.3. TASM32.exe

12.2. Основные отличия архитектуры процессоров i386/i486/Pentium от i8086

12.2.1. Основные регистры процессора

У всех регистров архитектуры процессора **i386** и старше, которые *расширяют* диапазон 16-разрядных регистров младших моделей процессора, в начале имени появляется буква **E** (Expanded - расширенный). В целях совместимости с младшими моделями процессоров **i8086** и **i286** действительны и все прежние наименования регистров (см. п. 3.3).

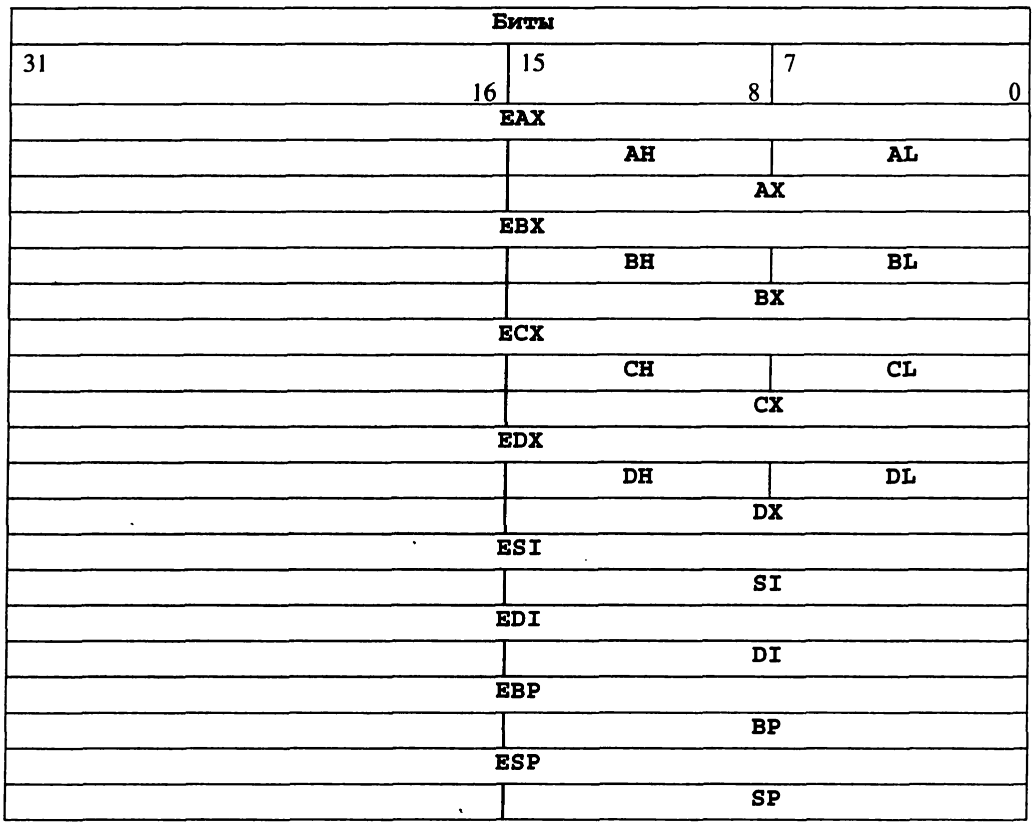
Регистров достаточно много. Мы остановимся на основных регистрах, чтобы немного уловить тенденцию развития семейства процессоров **Intel**. Для начала нам и этого будет более чем достаточно.

12.2.1.1. Регистры общего назначения

Основное отличие от аналогичных регистров процессора **i8086** заключается в том, что они стали 32-разрядными (отсюда и термин "*32-разрядное программирование*"). Кроме того, их стало восемь вместо четырех (см. рис 12.1).

12.2.1.2. Сегментные регистры — селекторы

Длина сегментных регистров НЕ изменилась, они по-прежнему 16-разрядные, но к четырем уже известным нам (см. п. 3.3.2) добавились еще два регистра **FS** и **GS** для дополнительных сегментов данных. Другое название этих регистров - селекторы связано с их специальным использованием в защищенном режиме (см. п. 12.3.2).



Биты		
31 … 16	15 … 8	7 … 0
EAX		
	AH	AL
	AX	
EBX		
	BH	BL
	BX	
ECX		
	CH	CL
	CX	
EDX		
	DH	DL
	DX	
ESI		
	SI	
EDI		
	DI	
EBP		
	BP	
ESP		
	SP	

РИС. 12.1. *Регистры общего назначения процессора i386.*

12.2.1.3. Регистр указателя команд

Этот регистр стал 32-разрядным и, соответственно, носит название **EIP**. Назначение его осталось прежним (см. п. 3.3.3).

12.2.1.4. Регистр флагов

Этот регистр стал называться **EFLAGS** и при своем расширении получил новые биты, которые мы и рассмотрим (см. рис. 12.2). Назначение прежних битов осталось БЕЗ изменения (см. п. 3.3.4).

Регистр флагов (EFlags)														
Биты														
31	…	22	21	20	19	18	17	16	15	14	13	12	…	0
	…		ID	VIP	VIF	AC	VM	RF		NT	IOPL		…	CF

РИС. 12.2. *Регистр флагов процессора i386/i486/Pentium.*

Бит	Назначение
IOPL (Input/Output Privilege Level)	Уровень привилегий ввода/вывода. Содержит два бита, которые показывают максимальную величину уровня привилегий ввода/вывода для текущей задачи. Используется в механизме защиты для управления доступом к адресному пространству ввода/вывода. Текущий уровень привилегий (CPL – Current Privilege Level) должен быть равен 0.
NT (Nested Task)	Вложенная задача. Если NT=1, то текущая задача считается вложенной по отношению к другой задаче. Некорректные изменения данного флага способствуют возникновению различных особых ситуаций в прикладных программах.
RF (Resumption Flag)	Флаг возобновления. Если RF=1, то временно отключается обработка исключительных ситуаций. Это делается для того, чтобы команда, вызвавшая особую ситуацию, могла быть перезапущена и не стала вновь причиной особой ситуации. Этот флаг используется совместно с отладочными регистрами или в пошаговом режиме.
VM (Virtual Mode)	Виртуальный режим (VM=1) в защищенном режиме процессора.
AC (Alignment Control) – i486	Контроль выравнивания. Флаг используется совместно с битом AM регистра CR0 (см. табл. 12.3) для отслеживания особых ситуаций, связанных с выравниванием операндов при обращении к оперативной памяти. Особая ситуация контроля выравнивания генерируется только, если уровень привилегий IOPL=3.
VIF (Virtual Interruption Flag) - Pentium	Виртуальное прерывание. Виртуальное подобие флага IF. Этот флаг применяется совместно с флагом VIP для обеспечения нормального функционирования устаревшего программного обеспечения, использующего команды управления маскируемыми прерываниями.
VIP (Virtual Interruption) – Pentium	Ожидание виртуального прерывания.
ID (Identification) – Pentium	Флаг идентификации. Проверка способности программы поддерживать команду идентификации процессора CPUID.

12.2.1.5. Управляющие регистры

Эти регистры (Control Registers) имеют длину 32 бита и имеют номера от 0 до 4: **CR0** (расширенный потомок управляющего регистра процессора **i286**), **CR1** (пока НЕ используется и является резервом фирмы **Intel**), **CR2**, **CR3**, **CR4** (появился в процессорах **Pentium**). Доступ к ним осуществляют специальные команды соответствующих процессоров.

Практически все режимы и специальные возможности процессора устанавливаются определенными битами в управляющих регистрах. Посмотрим, что же они могут.

Биты																	
31	30	29	28	...	19	18	17	16	15	...	6	5	4	3	2	1	0
PG	**CD**					**AM**		**WP**				**NE**	**ET**	**TS**	**EM**	**MP**	**PE**
									Слово состояния машины – MSW (i286)								

РИС. 12.3. *Формат регистра CR0.*

Таблица 12.3. Назначение битов регистра CR0.

Бит	Назначение
PE (Protection Enable) – i286	Разрешение защищенного режима. Если PE=0&PG=0, то процессор работает в реальном режиме.
MP (Monitor Soprocessor) – i286	Слежение за сопроцессором. Этот бит используется совместно с битом TS. Если MP=1&TS=1, то при выполнении команды WAIT генерируется особая ситуация 7 - «сопроцессор отсутствует». Подробнее о сопроцессорах – см. главу 13.
EM (Emulate Soprocessor) – i286	Эмуляция сопроцессора. Если EM=1, то генерируется особая ситуация 7 - «сопроцессор отсутствует» и все команды сопроцессора эмулируются. **При выполнении команд MMX этот флаг должен быть сброшен (EM=0).**
TS (Task Switched) – i286	Задача переключена. Этот бит устанавливается при каждом переключении задач и вызывает особую ситуацию 7 при поступлении команд сопроцессора, MMX/3DNow! и SIMD (см. табл. 12.8). Это нужно, чтобы предотвратить выполнение этих команд над данными, которые остались в оперативной памяти от другой задачи.
ET (Processor Extension Type) – i286, i486SX	Тип сопроцессора. Если ET=1, то используется 32-битный протокол сопроцессора i387. Если ET=0, то – 16-битный протокол i287. **Для процессоров Pentium ET=1.**
NE (Numeric Error) –I486	Ошибка сопроцессора. Если NE=0, то обработка исключения сопроцессора происходит в стиле MS DOS (через вызов внешнего прерывания). Если NE=1, то действует внутренний механизм обработки исключений при поступлении команд сопроцессора, MMX/3DNow! и SIMD.
WP (Write Protection) – i486	Защита записи. Если WP=1, то происходит защита от записи супервизором операционной системы страниц пользовательского уровня. В противном случае, - такая защита снимается.
AM (Alignment Mask) – i486	Маска выравнивания. Контроль выравнивания операндов в оперативной памяти выполняется, если AM=1&AC=1, IOPL =3 (см. табл. 12.2). Для i386 AM=0.
NW (No Write) – i486	Запрет сквозной записи. Работает вместе с битом CD и предназначен для управления внутренней кэш-памятью.
CD (Cash Data) – i486	Разрешение работы внутренней кэш-памяти (L1,L2) - CD=0.
PG(Paging Enable) –i286	Включение режима страничной адресации (PG=1).

Регистр **CR2** содержит полный 32-разрядный линейный адрес при возникновении особой ситуации 14 "страничная ошибка", если в регистре **CR0** бит **PG=1.**

Биты																	
31	30	29	28	...	12	11	10	9	8	7	6	5	4	3	2	1	0
20 старших бит физического адреса каталога страниц													PCD	PWT			

РИС. 12.4. *Формат регистра CR3.*

Таблица 12.4. Назначение битов регистра CR3.

Бит	Назначение
PWT – i486	Сквозная запись страниц. Используется для управления кэшированием текущего каталога страниц, если установлен режим страничной адресации (бит PG=1 в регистре CR0). Если PWT=1, то обновление текущего каталога страниц происходит методом сквозной записи, иначе – методом обратной записи.
PCD – i486	Запрещение кэширования страниц (PCD=1).

Дополнительные возможности по включению/выключению самых разнообразных возможностей процессоров семейства Pentium обеспечивает регистр **CR4**. Посмотрим, что в нем предусмотрено вплоть до процессоров Pentium-III (см. рис. 12.5 и табл. 12.5).

Биты													
31	...	11	10	9	8	7	6	5	4	3	2	1	0
			OSXMMEXCPT	OCFXSR	PCE	PGE	MCE	PAE	PSE	DE	TSD	PVI	VME

РИС. 12.5. *Формат регистра CR4 (Pentium).*

12.2.1.6. Регистры системных адресов

Это очень важные регистры и используются в защищенном режиме работы процессора, еще начиная с **i286** (см. п. 12.3.2). Регистры **GDTR** (Global Descriptors Table Register), **LDTR** (Local Descriptors Table Register) и **IDTR** (Interrupt Descriptors Table Register) предназначены для хранения базовых адресов таблиц дескрипторов (соответственно: глобальных, локальных и прерываний). Есть еще регистр задачи **TR** (Task Register), который применяется для получения информации, какая задача выполняется процессором в текущий момент.

Регистры **GDTR** и **IDTR** (см. рис. 12.6) содержат 32-битный (24-битный для **i286**) базовый адрес и 16-битный предел таблицы глобального дескриптора и таблицы дескриптора прерываний соответственно.

Биты		
47 46 ... 17 16	15 14 ... 1 0	Регистр
32-битный линейный базовый адрес	Предел	**GDTR** / **IDTR**

РИС. 12.6. *Формат регистров GDTR и IDTR.*

Таблица 12.5. Назначение битов регистра CR4.

Бит	Назначение
VME	Расширение виртуального V86 режима. Включается (VME=1) поддержка флага виртуальных прерываний VIF.
PVI	Виртуальные прерывания для защищенного режима. Включается (PVI=1) поддержка флага виртуальных прерываний VIF.
TSD	Ограничение маркера времени. Если TSD=0, то выполнение привилегированной команды **RDTSC** разрешается на всех уровнях привилегий. Иначе, только на самом высоком, системном, уровне – 0.
DE	Расширение отладки. Разрешаются (DE=1) точки останова по вводу/выводу.
PSE	Расширение размера страниц. Если PSE=1, то можно использовать страницы расширенного размера (4M или 2M байта в зависимости от текущей разрядности физического адреса). В противном случае размер страниц ограничен 4K байт.
PAE – Pentium Pro	Расширение физического адреса. Если PAE=1, то разрешается использовать 36-битную физическую адресацию (вместо 32-битной). При этом логический адрес остается 32-битным, а изменения относятся только к работе страничного механизма.
MCE	Расширение контроля работы процессора (MCE=1).
PGE – Pentium Pro	Разрешение (PGE=1) использования глобальных страниц – нужно для системных процедур.
PCE – Pentium Pro	Разрешение (PCE=1) использовать в пользовательских программах счетчика производительности процессора (привилегированная команда **RDPMC**).
OCFXSR – Pentium-II	Разрешение (OCFXSR=1) использовать команды быстрого сохранения/восстановления состояния сопроцессора/MMX/SIMD – команды **FXSAVE/FXSTOR** – см. табл. 13.14.
OSXMMEXCPT – Pentium-III	Разрешение особой ситуации SIMD.

Регистры **LDTR** и **TR** имеют длину 10 байт, но видимой частью (которую можно программно читать и изменять) являются только 16 бит, где содержатся селектор дескриптора локальной таблицы дескрипторов (**LDT**) и селектор дескриптора сегмента состояния задачи (**TSS**) соответственно. Сами дескрипторы **LDT** и **TSS** (базовый адрес, размер, атрибуты) автоматически загружаются в скрытую часть своих регистров из глобальной таблицы дескрипторов.

12.2.2. Логический адрес. Формирование эффективного адреса

Команды, как мы знаем, могут работать с регистрами, константами и с операндами, находящимися в оперативной памяти, — логическими адресами. До сих пор мы знали, что он формируется из двух величин:

<сегментный регистр>:<16-разрядный эффективный адрес>

16-разрядный эффективный адрес позволял делать различные режимы адресации (см. табл.8.3 и табл. 12.6) в довольно узком диапазоне.

А что происходит при 32-разрядном программировании? Логический (или он еще называется *виртуальным*) адрес получается аналогично приведенному выше и имеет следующий формат:

<селектор сегмента>:<32-разрядный эффективный адрес>

Эффективный 32-разрядный адрес (EA) в общем случае вычисляется сложением любой комбинации следующих четырех адресных элементов:

EA = База + (Индекс * Масштаб) + Смещение,

где

- **Смещение** — 8-, 16- или 32-разрядное значение.
- **База** (Base) — содержимое любого из 32-разрядных регистров общего назначения.
- **Индекс** (Index) — содержимое любого из 32-разрядных регистров общего назначения, кроме **ESP**.
- **Масштаб** (Scale) — константа 2, 4 или 8, на которую умножается значение индексного регистра.

Таблица 12.6.
Разница в использовании регистров и смещений в 16- и 32-разрядном программировании.

Компоненты EA	16-разрядный адрес	32-разрядный адрес
Базовый регистр	BX,BP	Любой 32-разрядный регистр общего назначения
Индексный регистр	SI,DI	Любой 32-разрядный регистр общего назначения, кроме ESP
Масштаб	НЕТ	2, 4, 8
Смещение	0, 8, 16 бит	0, 8, 16, 32 бита

Чтобы отразить информацию об адресации в 32-разрядном программировании, в машинном коде появляется байт **SIB** (расширение байта способа адресации **mod reg r/m**):

Биты							
7	6	5	4	3	2	1	0
S		I			B		

РИС. 12.7. *Формат байта SIB.*

Таблица 12.7.
Кодировка поля S(Scale).

S	Масштаб
00	Не используется
01	2
10	4
11	8

Значения индексного регистра (**I**) и регистра базы (**B**) кодируются, согласно табл. 12.8.

Таблица 12.8. Значения полей Index и Base.

Код	Поле I (Index)	Поле B (Base)
000	**EAX**	**EAX**
001	**ECX**	**ECX**
010	**EDX**	**EDX**
011	**EBX**	**EBX**
100	-	**ESP**
101	**EBP**	**EBP**, только если **S=01**, **S=10** или **S=11**
110	**ESI**	**ESI**
111	**EDI**	**EDI**



ЗАМЕЧАНИЯ

1) Если программа оперирует одновременно с 16-разрядными и 32-разрядными **регистрами**, то для их различения Turbo Assembler автоматически добавляет в объектный (машинный) код программы **префикс перекрытия размера операнда 66h** (opSize Prefix)— см. табл. П7.1.

2) Если программа оперирует одновременно с 16-разрядными и 32-разрядными **адресами**, то для их различения Turbo Assembler автоматически добавляет в объектный код программы **префикс перекрытия размера адреса 67h** (AddrSize Prefix) — см. табл. П7.1 и п.12.5.

3) Если в команде индексная адресация НЕ используется, то поле **S** игнорируется, а поле **I=reg**.

Сегментные регистры дополнились еще двумя регистрами-селекторами (см. п. 12.2.1.2), поэтому немного изменится рис 8.8 и табл. 8.2.

Биты	7	6	5	4	3	2	1	0
	mod		regS			r/m		

РИС. 12. 8. *Байт способа адресации для команд, работающих с селекторами.*

Таблица 12.9. Кодировка селекторов.

regS	Селекторы
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS

Вот и все изменения в машинном коде команды в 32-разрядном программировании! Подробнее о машинном коде см. п. 12.5.

12.2.3. Режимы адресации

Процессор **i386** поддерживает **11 режимов адресации**. Некоторые из этих режимов нам уже знакомы (см. главу 8). Сравним и посмотрим, что же изменилось.

1. Регистровый режим адресации.

Здесь все достаточно просто. При 32-разрядном программировании 16-битные регистры заменяются на 32-битные: AX — EAX, BX — EBX, ... , DI — EDI.

При этом и 8-разрядные, и 16-разрядные регистры тоже имеют право на существование. Например,

```
MOV AL,BH     ;  8-разрядный регистровый режим адресации
MOV AX,BX     ; 16-разрядный регистровый режим адресации
MOV EAX,EBX   ; 32-разрядный регистровый режим адресации
```

Машинный код последней команды будет аналогичен коду предыдущей команды **MOV AX,BX**. Но перед байтом кода операции в 32-разрядной (или 16-разрядной) команде может появиться **префикс перекрытия размера операнда 66h** — см. п. 12.5.

2. Непосредственный режим адресации.

Это — нам тоже знакомо. В команду непосредственно встраивается константа. Например:

```
ADD    AH,   6
ADD    AX,  12345
ADD    EAX,123456
```

3. Режим прямой адресации.

Адрес операнда задается в виде 8-, 16- или 32-битового смещения в самой команде, например,

```
DEC        WORD   PTR  h[500]
```

; аналог предыдущей команды:

```
DEC     WORD   PTR h+500
INC        DWORD PTR  a
```

4. Режим регистровой косвенной адресации.

В этом случае базовый или индексный регистр содержат адрес операнда. Например,

```
SUB        DX,  [SI]
SUB        EDX,[ECX]
```

5. Базовая адресация.

Содержимое базового регистра суммируется со смещением. Например,

```
ADD        DX,  [BX]+100
SUB        EDX,[EAX]+10
```

6. Индексная адресация.

Содержимое индексного регистра суммируется со смещением. Например,

```
ADD        DX,  brr[BP]
SUB        EAX,arr[EAX]
```

7. Индексная адресация с масштабированием.

Содержимое индексного регистра умножается на масштабный коэффициент и суммируется со смещением. Например,

```
SUB        EAX,arr[EAX*4]
```

8. Базово-индексная адресация.

Содержимое базового регистра складывается с содержимым индексного регистра. Например,

```
ADD       EAX, [EBX+ESI]
```

9. Базово-индексная адресация с масштабированием.

Содержимое индексного регистра умножается на масштабный коэффициент, и результат суммируется с содержимым базового регистра. Например,

```
SUB       ECX, [EBX+EAX*8]
```

10. Базово-индексная адресация со смещением.

Содержимое базового регистра складывается с содержимым индексного регистра и со смещением. Например,

```
SUB       EDX, ECX[EAX+10]
ADD       EAX, [ESI][EBP+00ABCFh]
```

; аналог предыдущей команды:

```
ADD       EAX, [ESI+EBP+00ABCFh]
```

11. Базово-индексная адресация со смещением и масштабированием.

Содержимое индексного регистра умножается на масштабный коэффициент, и результат добавляется к содержимому базового регистра, которое суммируется со смещением. Например,

```
SUB       ECX, table[EBX+36+EAX*8]
```

; аналог предыдущей команды:

```
SUB       ECX, table[EAX*8][EBX+36]
```

12.2.4. Дополнительные типы данных

Процессор **i386** и все последующие модели поддерживают все рассмотренные нами ранее типы данных (см. табл. П4.1 и П4.3) и вводит новые (сравните с табл. П4.2 и П4.4).

1. **Счетверенное слово без знака** — в формате 64 бита.
2. **Счетверенное слово со знаком** — в формате 64 бита.
3. **Битовое поле** — группа, включающая до 32-х смежных бит.
4. **Битовая строка** — множество смежных битов длиной до 4Г байт.
5. **Короткий указатель** — 32-битовое смещение относительно базового адреса сегмента, содержащегося в селекторе сегмента. Адресация возможна в пределах одного сегмента (до 4Г байт).
6. **Длинный указатель** — полный указатель вида:

<селектор сегмента>:<32-разрядный эффективный адрес>.

7. **Строка** — последовательность смежных байтов, слов или двойных слов длиной до 4Гбайт.

12.2.5. Организация оперативной памяти и адресного пространства

Кроме типов данных, указанных в п. 12.2.4, процессор **i386** поддерживает еще два типа больших блоков памяти: *страницы* и *сегменты*.

Сегментная организация памяти. Память можно разбить на один или несколько сегментов различной длины (до 4Г байт), которые могут подкачиваться с диска по мере необходимости и использоваться различными программами. Это удобно при разработке прикладных задач, работающих с логическими адресами. Поскольку каждая задача может работать с 16K селекторов, а смещения могут адресовать 4Г байт, то виртуальное адресное пространство, доступное программе, составляет 64Т байт (1Т байт=1024Г байт).

Страничная организация памяти. Вся память представляется в виде совокупности страниц размером по 4K байт. Каждая из этих страниц обладает набором свойств, определяющих правила доступа к ней, а также положение этой страницы либо в физическом адресном поле оперативной памяти, либо на внешнем устройстве. Всякий раз, когда происходит обращение к конкретной странице (по линейному адресу), процессор автоматически определяет соответствующий физический адрес. Этот метод удобен при решении задач системного характера, поскольку позволяет в условиях дефицита физической оперативной памяти организовать виртуальную память на внешних носителях информации, например, на винчестере. Страничный механизм организации памяти может быть задействован только в защищенном (в том числе и в V86) режиме работы процессора (см. п. 12.3.2 и п.12.3.3). **Для процессоров Pentium размер страницы стал больше — 4M байт.**

Процессор **i386** имеет три различных адресных пространства: логическое, линейное и физическое. *Логический (виртуальный) адрес* мы рассматривали в п. 12.2.2. При отключенном страничном механизме организации памяти *линейный адрес* полностью совпадает с физическим. Способ его формирования зависит от текущего режима работы процессора (реальный, защищенный, 32- или 16-битная адресация).

Физический адрес передается на внешнюю шину для обращения к ячейкам памяти. Процессоры **Pentium** позволяют адресовать до 2^{36} байт (64Г байт) в случае поддержки механизма расширения физического адреса (см. флаг **PAE** регистра **CR4** — табл. 12.5).

12.2.6. Прерывания и особые ситуации

Прерывания и особые ситуации, изменяют нормальное выполнение программы. Различие между прерываниями и особыми ситуациями состоит в том, что *прерывания* обрабатывают внешние события (например, ввод/вывод информации), а *особые ситуации* являются реакцией на ошибки, возникающие при выполнении команд.

Аппаратные прерывания бывают двух видов: маскируемые (обычные) и немаскируемые (системные — с очень высоким приоритетом, например, сбой питания). Обработка маскируемых прерываний (если они разрешены, — бит **IF=1** регистра **EFLAGS**) выполняется после выполнения текущей команды. После того, как программа обработки прерываний закончит обслуживание прерывания, выполнение программы продолжится со следующей команды — подробнее см. главу 14.

Особые ситуации могут генерироваться:

- **процессором по результатам выполнения команд;**
- **программными прерываниями;**
- **средствами самопроверки процессора.**

Они разделяются на три типа: **ошибка, ловушка, сбой** и имеют свой номер (см. табл. 12.10).

Двойными ошибками называются случаи возникновения особой ситуации во время выполнения процессором обработки возникшей ранее особой ситуации. Большинство прерываний (кроме деления на ноль — прерывание 0) и сегментные ошибки (прерывания 10, 11, 12, 13) могут приводить к двойным ошибкам.

Таблица 12.10. Типы прерываний и особых ситуаций.

Название	Номер n	Источник возникновения	Процессор	Тип
Деление на нуль	0	Команды IDIV,DIV	8086	Ошибка
Прерывание отладки	1	Любые команды	8086	Ошибка /Ловушка
Немаскируемое прерывание	2		8086	Прерывание
Точка останова	3	Команда INT 3	8086	Ловушка
Переполнение	4	Команда INTO	8086	Ловушка
Нарушение границ	5	Команда BOUND	80186	Ошибка
Неопределенный код операции	6	Зарезервированные коды операций. Некорректные команды.	286	Ошибка
Сопроцессор отсутствует	7	Команды ESC, WAIT. Команды MMX, 3DNow!, SIMD – см. табл. 12.3	286	Ошибка
Двойная ошибка	8	Любые команды	286	Сбой
Перегрузка сегмента сопроцессора. Начиная с i486, зарезервировано.	9	Команды ESC, WAIT	Только 286,386	Ошибка
Неправильный TSS*)	10	Команды JMP, CALL, IRET. Прерывания и особые ситуации	286	Ошибка
Сегмент отсутствует*)	11	Любая команда, изменяющая сегмент	286	Ошибка
Ошибка стека	12	Стековые операции	286	Ошибка
Общая ошибка защиты	13	Любые неверные ссылки на код или данные	286	Ошибка /Ловушка
Страница отсутствует*)	14	Любые ссылки на код или данные	386	Ошибка
Зарезервировано	15			
Ошибка сопроцессора	16	Команды ESC, WAIT	286	Ошибка
Контроль выравнивания*)	17	Любые ссылки на данные	486	Ошибка
Контроль процессора	18	Зависит от модели	Pentium	Ошибка
SIMD-исключение	19	Команды SIMD	Pentium III	Ошибка
Зарезервировано	20-31			
Прерывания пользователя	32-255	Внешние прерывания или команды INT n	8086	Прерывание

* Особые ситуации генерятся только в защищенном режиме или в режиме V86

12.3. Режимы работы

В п. 3.4.3 был упомянут защищенный режим работы процессора, элементы которого появились впервые у процессоров **i286**. С появлением процессора **i386** начала осуществляться полная поддержка защищенного режима и появился новый, виртуальный режим **V86**.

Виртуальная память отличается от обычной оперативной памяти тем, что находится обычно на внешнем носителе информации, где хранятся редко используемые фрагменты кода, которые могут подгружаться в оперативную память по мере необходимости. Это позволяет снять ограничение, накладываемое на объем физической оперативной памяти.

Мультизадачность означает одновременное решение на одном процессоре нескольких задач. Например, современные операционные системы линейки Windows как раз и реализуют эту многозадачность в защищенном режиме работы процессора. Грамотная реализация мультизадачности при аварийном завершении одной из задач НЕ должна приводить к остановке других задач и всей операционной системы в целом.

12.3.1. Реальный режим

При включении питания или после сигнала **Reset** (Сброс) ВСЕ процессоры семейства **ix86** устанавливаются в реальный режим работы (Real Mode), который соответствует базовому процессору **i8086** с добавлением возможности использования 32-разрядных регистров, начиная с процессора **i386**. При этом механизм адресации, пространство адресов памяти, управление прерываниями осуществляются аналогично реальному режиму процессора **i8086** — см. п. 3.4.3 и прил. 6.

В реальном режиме могут использоваться любые команды. Размер операнда по умолчанию такой же, как и у процессора **i8086** (16 бит). Чтобы использовать 32-разрядные регистры и режимы адресации, в машинном коде появляется **префикс перекрытия размера операнда 66h.** Но размер сегмента в реальном режиме НЕ должен превышать 64К байт (см. 3.4.2), поэтому 32-битовые адреса должны быть меньше, чем **0000FFFFh**. Попытка использования смещения больше этой величины вызывает особую ситуацию 12 или 13.

В реальном режиме есть две зарезервированные области памяти: зона системной инициализации (адреса от **FFFFFFF0h** до **FFFFFFFFh**) и таблица векторов прерываний (адреса от **00000h** до **003FFh**).

Многие особые ситуации (например, 10,11,14,17) в реальном режиме НЕ используются (см. табл. 12.10). Некоторые особые ситуации имеют иной смысл — см. табл. 12.11.

В случае возникновения грубой ошибки может произойти останов процессора.

Основной функцией реального режима является подготовка современных процессоров к работе в защищенном режиме.

Таблица 12.11. Особые ситуации в реальном режиме.

Номер	Функция	Причина	Адрес возврата
8	Запрос несуществующего прерывания	Номер прерывания в команде INT превышает размер таблицы прерываний [0,...,255]	Предыдущая команда
13	Прерывание при превышении размера сегмента	Адресуемое слово имеет смещение 0FFFFh или выполняемая команда находится за границей сегмента	Предыдущая команда

12.3.2. Защищенный режим

В защищенном режиме (Protected Mode) адресное пространство расширяется до 4Г байт, а область виртуальных адресов — до 64Т байт. При этом используется другой механизм адресации.

Вся оперативная память разбивается на сегменты. Но в отличие, от реального режима, эти сегменты являются самостоятельными единицами памяти, имеющими собственные атрибуты: базовый адрес сегмента, размер, тип, привилегии доступа и проч. Вся информация о сегменте запоминается в таблице дескрипторов (указателей). Этих таблиц три:

- **GTD** (Global Descriptors Table) — таблица глобальных дескрипторов;
- **LTD** (Local Descriptors Table) — таблица локальных дескрипторов;
- **ITD** (Interrupt Descriptors Table) — таблица дескрипторов прерываний.

Эти таблицы создаются и заполняются ДО переключения в защищенный режим.

Для обращения к таблицам дескрипторов применяются регистры системных адресов (см. п. 12.2.1.6). Каждая из таблиц хранится в своей области памяти и имеет размер от 8 байт до 64К байт.

Таблица глобальных дескрипторов содержит дескрипторы, доступные для ВСЕХ задач системы. Обычно **GTD** содержит код и сегменты данных, используемые операционной системой, сегменты состояния задачи, а также дескрипторы таблиц **LTD**.

Таблица локальных дескрипторов содержит дескрипторы, используемые в какой-то одной задаче. В современных операционных системах принято, чтобы каждая задача имела собственную **LTD**. Этим обеспечивается механизм защиты каждой отдельной задачи и всей системы в целом.

Для формирования логического адреса, как и в реальном режиме, используются два компонента:

<селектор сегмента>:<32-разрядный эффективный адрес>

Селектор сегмента, в отличие от реального режима, содержит не базу физического адреса сегмента, а *указатель* (*дескриптор*) на описание сегмента в какой-либо таблице дескрипторов (**GTD** или **LTD**).

Биты															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Индекс													TI	RPL	

РИС. 12.9. *Формат селектора в защищенном режиме.*

Используя выбранные таким образом *дескриптор* и *32-разрядный эффективный адрес,* процессор вычисляет логический адрес, складывая базовый адрес (взятый из таблицы дескрипторов) с эффективным адресом. Если не используется механизм страничной адресации, то этот логический адрес является и физическим адресом. В противном случае он еще дополнительно преобразуется в физический адрес.

В защищенном режиме может использоваться и 16-разрядная модель адресации. Это задается в дескрипторе соответствующего сегмента кода. При этом эффективный адрес тоже должен быть 16-разрядным. Но методы формирования физического адреса через селекторы и дескрипторы остаются такими же, как и при 32-разрядной адресации.

Таблица 12.12. Назначение битов селектора.

Бит	Назначение
RPL (Requested Privilege Level)	Запрашиваемый уровень привилегий. Определяет уровень привилегий, запрашиваемых селектором (**RPL=0** - наивысший уровень привилегий). Процессор осуществляет проверку прав доступа. Если программа, имеющая более низкий приоритет (**RPL=3**), попытается получить доступ к сегменту памяти с более высоким приоритетом (2,1,0), то будет сгенерирована особая ситуация 13 – ошибка общей защиты.
TI (Table Indicator)	Бит указания типа таблицы дескрипторов: **TI=0 (GDT), TI=1 (LDT)**.
Индекс	Смещение внутри таблицы дескрипторов. Нужен для вычисления адреса нужного дескриптора.

12.3.3. Виртуальный режим V86

Это специальное подмножество защищенного режима. Режим **V86** позволяет создавать на базе компьютеров с процессором, начиная с **i386**, и многозадачной операционной системы многопользовательские вычислительные системы. При этом у каждого пользователя создается иллюзия монопольного владения ВСЕМИ ресурсами компьютера.

Реальный режим работы процессора **i386**, эмулирующий процессор **i8086**, и режим виртуального 8086 (**V86**), работающий в защищенном режиме **i386**, несколько отличаются за счет использования механизма страничной адресации.

- **Эффективный адрес** вычисляется так же, как и в реальном режиме (см. п. 3.4.3).
- **Пространство задачи в 1М байт** может быть размещено в любом месте 4Г байт пространства линейных адресов процессора **i386**. Адреса, превышающие 1М байт, приводят к возникновению особой ситуации 13 — ошибка общей защиты.

В компиляторах **Borland C++**, начиная с версии 4.0, именно этот режим реализован на так называемых платформах **Win16**. Кроме того, если делается проект для **DOS** и программа выполняется под управлением многозадачной операционной системы линейки **Windows**, то она тоже работает в режиме **V86**.

Таким образом, все наши примеры, которые мы сделали до сих пор, при условии их работы на процессорах, начиная с **i386**, и под управлением операционных систем **Win9x, NT 4, 2000, XP** выполняются в режиме **V86**. Если вы обращали внимание на даты, то примеры, которые младше 1993 года спокойно работали и в обычной MS DOS, начиная с процессора **i8086**, только модель была не **large**, а **small** (процессор **Pentium** НЕ захотел работать с моделью **small** — пришлось заменить на **large**). Есть и более древние примеры, которые вообще не ссылались на модель памяти — в них в явном виде указывались директивы **SEGMENT** (см. пример 5.5).

12.4. Особенности программирования в 32-х разрядном коде

Теперь попробуем решить задачи в родном для операционной системы линейки **Windows** *защищенном режиме*. Проиллюстрируем наши действия на примерах, реализованных, как в старых 16-разрядных средах программирования типа **Borland (Turbo) Pascal-7.0x**, так и в современных, 32-разрядных: **Borland Delphi-5** и **Borland C++ Builder-5**.

Все примеры были протестированы автором в операционных системах **Windows 95-OSR2.1**, **Windows NT 4 Server** (процессор **Pentium-90**) и **Windows 2000** (процессор **AMD Athlon-650**) и дали одинаковый результат. Замечательное это дело — преемственность процессоров семейства **ix86**!

12.4.1. Дополнительные команды

Рассмотрим некоторые дополнительные команды процессора **i386**, расширяющие возможности уже изученных нами команд и позволяющие работать с 32-разрядными данными.

12.4.1.1. Команды пересылки MOVSX и MOVZX

Команда **MOVSX** (MOVe and Sign eXtension) выполняет пересылку элементов меньшей размерности в эквивалентные им элементы большей размерности с учетом знака. Команда **MOVZX** (MOVe and Zero eXtension) делает то же самое, но для БЕЗЗНАКОВЫХ данных.

Синтаксис:

```
MOVSX    Приемник, Источник
MOVZX    Приемник, Источник
```

Примеры.

```
MOV      AL, 0Ah
MOVSX    BX,AL               ; BX=000Ah
MOVZX    DX,AL               ; DX=000Ah
MOV      AH, AAh
MOVSX    CX,AH               ; BX=FFAAh
MOVZX    ECX,AH              ; ECX=000000AAh
MOVZX    ESI, WORD PTR a
```

12.4.1.2. Команды работы со стеком PUSHAD, POPAD, PUSHFD и POPFD

Команда **PUSHAD** расширяет возможности команды **PUSHA** (см. п. 5.1.4.2) и запоминает все (ALL Double) регистры в стеке СТРОГО в следующей последовательности: **EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI**.

Команда **POPAD**, наоборот, извлекают все (ALL Double) эти регистры из стека: **EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX**.

Синтаксис:

```
PUSHAD
POPAD
```

Команды **PUSHFD** запоминают или восстанавливают (**POPFD**) регистр флагов **EFLAGS**, используя стек.

Синтаксис:

```
PUSHFD
POPFD
```

Эти две команды расширяют возможности команд **PUSHF** и **POPF** (см. п. 5.1.4.3). Как видите, все достаточно просто.

12.4.1.3. Команды распространения знака CDQ и CWDE

По аналогии с командами CWD и CBW (см. п. 5.2.5) для **ЗНАКОВЫХ данных** существуют две команды распространения знака. **CDQ** (Convert Double word to Quadr word — преобразовать двойное слово, находящееся в регистре EAX, в учетверенное — <EDX:EAX>) и **CWDE** (Convert Word to Double word Expand — преобразовать слово, находящееся в регистре AX, в двойное слово, размещаемое в регистре EAX). Операнды им тоже НЕ нужны.

Синтаксис:

```
CDQ
CWDE
```

12.4.1.4. Команда целочисленного умножения со знаком IMUL

Расширен синтаксис команды **IMUL** (см. п. 5.2.3):

```
IMUL    Множитель1, Множитель2
IMUL    Результат, Множитель1, Множитель2
```

Логика работы команд:

<*Множитель1*> = <*Множитель1*> * <*Множитель2* >

< *Результат*> = <*Множитель1*> * <*Множитель2* >

Операнды этой команды становятся явными и более гибкими, что значительно упрощает ее использование. Первый операнд является искомым произведением. Он может быть только регистром R16/R32. Остальные операнды могут быть константами, регистрами или областью в памяти (Im8/Im16/Im32/R16/R32/Mem16/Mem32). Их длина (кроме констант) должна соответствовать длине первого операнда. При этом автоматически (при необходимости) может быть сделано распространение знака, т.е. расширение операнда. Например, все ниже следующие команды имеют право на существование:

```
IMUL   EBX
IMUL   BX, d[BP+SI]
IMUL   ECX,EBX
IMUL   BX, x,50000
IMUL   EDI,h,12
IMUL   AX,1245
IMUL   EDX,222
IMUL   CX,BX,121
IMUL   ECX,EBX,1234561
IMUL   EAX, 100000
IMUL   EAX, [ESI][EBP+00ABCFh]
IMUL   ECX, htable[EBX+36+EAX*8]
```

12.4.1.5. Команды обработки строк MOVSD, CMPSD, SCASD,LODSD, STOSD

Эти команды расширяют соответствующие команды обработки строк для двойных слов (см. п. 11.1).

12.4.2. Пример реализации Borland (Turbo) Pascal-7.0x+Turbo Assembler 3.2

Сделаем слегка измененную версию **примера 7.2** для знаковых 32-разрядных данных (типа **longint**). Устраним неприятность, которая была в этом примере для отрицательных НЕЧЕТНЫХ чисел. Кроме того, будем следить на Ассемблере за переполнением при сдвиге влево.

При решении воспользуемся уже **известными** нам командами Ассемблера и **расширенными** 32-разрядными регистрами.

Исходный текст модуля Shift32.asm

```
      title shift32 CopyRight by Голубь Н.Г. 1993-1998
            ;y=8*x
            ;z=w/4
            model      Large,Pascal
            .386
            .data
            Extrn      x:Dword,w:Dword,y:Dword,z:Dword
            .code
            Public     shift32
shift32     proc       far
            mov        Eax,x
            mov        Ebx,w
            sal        Eax,3                 ; 8*x
            JO         Over
            mov        y,Eax
            TEST       Ebx,80000000h         ; знак?
            JNZ        Lab                   ; <0
            sar        Ebx,2                 ; w/4
Exit:       mov        z,Ebx
            ret
Lab:        neg        Ebx                   ; w:=-w
            sar        Ebx,2                 ; w/4
            neg        Ebx                   ; — результат
            JMP        Exit
Over:
            mov        z,80000000h
            ret
shift32     endp
            end
```

Реализация на Паскале (**Shift32.pas**) ничего интересного и нового нам НЕ несет — при желании можете посмотреть ее в прилагаемых к книге материалах.

Тестовые проверки

```
------ Test #1 ---------
Вычислить: y:=8*x;   z:=w/4;   x,w,y,z  :longint;
введите x,w:  111111111   111111111
x = 111111111;    w = 111111111
АССЕМБЛЕР:   y=888888888  z=27777777
ПАСКАЛЬ:     y=888888888  z=27777777

продолжать? (y/n)
```

```
------- Test #2 ---------
Вычислить: y:=8*x;  z:=w/4;  x,w,y,z :longint;
введите x,w:  -11111222  -2222222
x = -11111222;   w = -2222222

АССЕМБЛЕР:   y=-88889776  z=-555555
ПАСКАЛЬ:     y=-88889776 z=-555555

продолжать? (y/n)
------- Test #3 ---------
Вычислить: y:=8*x;  z:=w/4;  x,w,y,z :longint;
введите x,w:  12345678  87654321
x = 12345678;   w = 87654321

АССЕМБЛЕР:   y=98765424  z=21913580
ПАСКАЛЬ:     y=98765424 z=21913580

продолжать? (y/n)
------- Test #4 ---------
Вычислить: y:=8*x;  z:=w/4;  x,w,y,z :longint;
введите x,w:  1234567890  -1234567890
x = 1234567890;   w = -1234567890
!!!!!!!!!! ПЕРЕПОЛНЕНИЕ !!!!!!!!!!

ПАСКАЛЬ:   y= 9.87654312000000E+0009 z=-3.08641972500000E+0008

продолжать? (y/n)
------- Test #5 ---------
Вычислить: y:=8*x;  z:=w/4;  x,w,y,z :longint;
введите x,w:  -1234567890  1234567890
x = -1234567890;   w = 1234567890
!!!!!!!!!! ПЕРЕПОЛНЕНИЕ !!!!!!!!!!

ПАСКАЛЬ:   y=-9.87654312000000E+0009 z= 3.08641972500000E+0008

продолжать? (y/n)
------- Test #6 ---------
Вычислить: y:=8*x;  z:=w/4;  x,w,y,z :longint;
введите x,w:  123456789  -1234567890
x = 123456789;   w = -1234567890

АССЕМБЛЕР:   y=987654312  z=-308641972
ПАСКАЛЬ:     y=987654312 z=-308641972

продолжать? (y/n)
```

Вопросы

1. Действительно ли наша задача работает в *защищенном режиме*?
2. Как это определить?
3. От чего это зависит?

Подсказки.

1. Посмотрите на файл листинга **Shift32.lst.**
2. Если компиляция идет с использованием среды **Borland Pascal**, то можно задавать режимы компиляции программы (см. **Compile/Target**).

12.4.3. Пример реализации Borland Delphi-5.0+Turbo Assembler 5.3

Решим простейшую задачу: **x=a-b** для знаковых 32-разрядных (типа **integer**) данных. Воспользуемся родной 32-разрядной средой программирования Delphi-5.0. Сделаем проект **Console Application**, поскольку интерфейс операционной системы Windows со своими кнопочками и прочими прибамбасами нам пока НЕ нужен. Умная среда разработки создаст нам *пустой проект* (выделено *курсивом*), который мы заполним своим содержанием.

Исходный текст файла проекта SubLong.dpr

```
program SubLong;
{$APPTYPE CONSOLE}
uses SysUtils;
{———————————————————————————————————————————————
              x=a-b;  a,b,x : Integer; (Win32!!!)
              Лекционный пример  (Delphi -5)
              CopyRight by Голубь Н.Г., 1993-1997,2001
———————————————————————————————————————————————}
{$I subaL}    {вызов ASM-модуля subaL.obj}
Const
   {Символьные константы}
  inv1='Repeat Enter. Exit — Ctrl/C';
  inv2='Result ';
  inv3='is out RANGE [';
  inv4='Entered Value ';
  invA='A (Integer)';
  invB='B (Integer)';
  {Допустимый диапазон для вводимых данных и результата}
  LongMin=-2147483647-1;
  LongMax= 2147483647;
Label L1;
var
  a,b,x,xa  : Integer;
  ch        : Char;
  t         : byte;
Procedure suba(Var x:Integer; a,b:Integer); FAR; external; {FAR обязателен!!!!}
function F (a,b:Integer; Var x:Integer) : Boolean;
Var x1     : Real;
    s1     : String;
Begin
  F:=True;  {Ошибок НЕТ}
  {ВНИМАНИЕ !!!!!!!!!!! Принципиально важно для корректности вычислений !!!!!!!!!}
  x1:=a;
  x1:=x1-b;
  if (x1>=LongMin)and(x1<=LongMax) then x:=trunc(x1)
   else
     Begin
       s1:=inv2+inv3;  {Формирование сообщения об ошибке}
      Writeln(s1,LongMin,'..',LongMax,']!!!!');
       {Вывод полученного значения, выходящего ЗА ДОПУСТИМЫЙ диапазон}
       Writeln(x1);
       Writeln(inv1);
       F:=False; {Признак ошибки}
      Exit;
     End
End;
```

```
{ Процедура ввода целочисленных данных с проверкой на ДОПУСТИМЫЙ диапазон:
    A                     — результат КОРРЕКТНОГО ввода
    inv                   — строка приглашения на ввод переменной
    LongMin,LongMax       — ДОПУСТИМЫЙ диапазон
}
Procedure Input(Var A:Integer; inv:String; LongMin,LongMax:Integer);
Label L;
Var     aL     : Real;
        s1     : String;
Begin
  L:
  Repeat
    Write('??? ',inv,'===>');
   {Контроль ввода НЕ числовых символов}
   {$I-}
    Readln(aL);      {Промежуточный буфер ввода для ЦЕЛОЧИСЛЕННЫХ данных}
   {$I+}
  Until (IOResult=0);
  if (aL>=LongMin)and(aL<=LongMax) then  A:=trunc(aL)
   else
     Begin
       s1:=inv4+inv3;
      Writeln(s1,LongMin,'..',LongMax,']!!!!');
       Writeln(inv1);
       goto L;
     End
End;
begin
 // Insert user code here
  t:=0;
  Repeat
    inc(t);
    writeln('————————— Test #',t,' ——————————');
    Writeln('      x = a - b; a,b,x : Integer;');
    L1:
    Input(a,invA,LongMin,LongMax);
    Input(b,invB,LongMin,LongMax);
    if F(a,b,x) then Writeln ('Pascal:    x=',x)
      else goto L1; {Результат выходит за диапазон, повторяем ввод данных}
    // Assembler
    suba(a,b,xa); // Обратите внимание на ПОСЛЕДОВАТЕЛЬНОСТЬ параметров!!!
    writeln ('Assembler: x=',xa);
    Writeln('To be continue? (y/n)');
    Readln(ch);
  Until (ch='n') or (ch='N');
end.
```

Как видите, содержание **СОВСЕМ НЕ ИЗМЕНИЛОСЬ** по сравнению с тем, какое было в обычном Паскале — см. п. 5.2.1.4.

Теперь займемся модулем на Ассемблере. Компилятор тоже должен быть 32-разрядный — **tasm32.exe** (версия 5.x). Модель **large** при стыковке с файлом **SubLong.dpr** будет отвергнута. И, действительно, нужна **НОВАЯ МОДЕЛЬ — FLAT** (плоский). Эта именно модель и используется в **Win32-приложениях** — см. п. 3.4.2. С параметрами работать так, как мы привыкли в 16-разрядном программировании, тоже НЕ получается — компилятор НЕ принимает. Они должны передаваться в сегменте кода как ГЛОБАЛЬНЫЕ переменные. Зато использовать их в Паскале мы сможем так, как нам заблагорассудится. С одной стороны это удобно, а с другой — может привести к неприятным последствиям (как и всякая неопределенность).

Исходный текст модуля Subal.asm

```
    title subaL
               ;x=a-b      !!!!!!!! tasm32.exe SUBAL.ASM /l
               .386
               .MODEL flat,PASCAL
               .Code
; передаются как ГЛОБАЛЬНЫЕ параметры!!!!!!!!
               Extrn xa:Dword, a:Dword,b:Dword
               Public     suba
               suba       proc
;ARG           a:Dword,b:Dword  RETURNS  xa:Dword ??????????
               mov        Eax,a
               mov        Ebx,b
               sub        Eax,Ebx
               mov        xa,Eax
               ret
suba           endp
        end
```

Тестовые примеры

```
------- Test #1 ---------
x = a - b; a,b,x : Integer;
??? A (Integer)===>11111111
??? B (Integer)===>-1111111

Pascal:      x=12222222
Assembler:   x=12222222

To be continue? (y/n)

------- Test #2 ---------
x = a - b; a,b,x : Integer;
??? A (Integer)===>11111111111111
Entered Value is out RANGE [-2147483648..2147483647]!!!!
Repeat Enter. Exit - Ctrl/C
??? A (Integer)===>111111
??? B (Integer)===>4444444

Pascal:      x=-4333333
Assembler:   x=-4333333

To be continue? (y/n)
```

```
-------- Test #3 ----------
x = a - b; a,b,x : Integer;
??? A (Integer)===>-111111111
??? B (Integer)===>3333333333
Entered Value is out RANGE [-2147483648..2147483647]!!!!
Repeat Enter. Exit - Ctrl/C
??? B (Integer)===>333333333

Pascal:      x=-444444444
Assembler:   x=-444444444

To be continue? (y/n)
-------- Test #4 ----------
x = a - b; a,b,x : Integer;
??? A (Integer)===>-21111111111
Entered Value is out RANGE [-2147483648..2147483647]!!!!
Repeat Enter. Exit - Ctrl/C
??? A (Integer)===>-2111111111
??? B (Integer)===> 2111111111
Result is out RANGE [-2147483648..2147483647]!!!!
-4.2222222200000E+0009
Repeat Enter. Exit - Ctrl/C
??? A (Integer)===>-2111111111
??? B (Integer)===> 211111111
Result is out RANGE [-2147483648..2147483647]!!!!
-2.3222222200000E+0009
Repeat Enter. Exit - Ctrl/C
??? A (Integer)===> 211111111
??? B (Integer)===>-211111111

Pascal:      x=422222222
Assembler:   x=422222222

To be continue? (y/n)
```

Тестовые примеры работают нормально.

12.4.4. Пример реализации Borland C++ Builder-5.0+Turbo Assembler 5.3

Теперь реализуем в 32-разрядном режиме немного более сложный **пример 6.2**, где входная переменная **a** будет проверяться на вводе как 16-разрядное целое знаковое число, а передаваться в Ассемблер как 32-разрядное число типа **int**. Результат **X** тоже пусть будет типа **int.** Изменим немного и константы **b, c, d**.

Новых команд здесь, в общем, НЕТ. Изменены только 16-разрядные регистры на расширенные. Для облегчения жизни процессору добавлены команды загрузки в стек всех регистров и флагов с последующим их восстановлением по окончании работы модуля.

Исходный текст модуля PRIM_32.asm

```
        title Арифметические выражения
        .386
        model   FLAT,C
; tasm32.exe PRIM.ASM /l /ml !!!!!!!!!!!!!!!
; CopyRight by Голубь Н.Г., 1993-1997, 2000-2001
; Пример 6.2. Win32
;x=(2*a + b*c)/(d-a), d<>a !!!
;int x,a,b,c,d;
```

```
             .CODE
;============= ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
             Extrn      C X:Dword
             Extrn      C a:Dword
;=========== ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ - КОНСТАНТЫ
b            dD         -333333
c            dD         1000
d            dD         -10
;==================================================
             Public     C prim
prim         proc
             pushAD
             pushFD
             mov        Eax,2
             Imul       a         ; <Edx>:<Eax>=2*a
             mov        Ebx,Edx   ; Ebx <=== ст.часть (Edx)
             mov        Ecx,Eax   ; Ecx <=== мл.часть (Eax)
             mov        Eax,b
             Imul       c         ; <Edx>:<Eax>=b*c
             add        Eax,Ecx   ; <Eax>=<Eax>+<Ecx>(мл.часть)
             adc        Edx,Ebx   ; <Edx>=<Edx>+<Ebx>(ст.часть)
             mov        Ecx,d
             sub        Ecx,a     ; <Ecx>=<Ecx>-a
             Idiv       Ecx       ; <Eax>=<Edx>:<Eax>/<Ecx>
             mov        X,Eax
             popFD
             popAD
             ret
prim         endp
         end
```

Программу на языке C++ изменять НЕ будем. Откомпилируем ее и **asm**-файл с помощью 32-разрядного компилятора командной строки **bcc32.exe**, создав командный файл **Make.bat**, аналогичный приведенному в п. 6.1.2.5. Изменены будут только следующие строки:

```
set  path=%path%;N:\CBuilder5\bin;N:\CBuilder5\lib;N:\TASM5\BIN
set  include=N:\CBuilder5\INCLUDE
set  lib=N:\CBuilder5\LIB
set  link=N:\CBuilder5\BIN
tasm32 %2 temp.obj /l /ml
bcc32 %1 temp.obj
```

Полностью файл **Make.bat** можно посмотреть в прилагаемых к книге материалах.



ПРИМЕЧАНИЕ.

Компилятор **tasm32.exe версии 5.3** входит в стандартную поставку **Borland C++ Builder-5.0**.

Тестовые примеры

```
================ Test #1 ================
      x=(2*a+b*c)/(d-a);   int x,a,b=-333333,c=1000,d=-10;
      Enter a [-32768..32767], a!= -10 ====> 1
  Result (C++)     x = 30302999
  Result (ASM)     x = 30302999
Exit? - (y/n)
================ Test #2 ================
      x=(2*a+b*c)/(d-a);   int x,a,b=-333333,c=1000,d=-10;
      Enter a [-32768..32767], a!= -10 ====> -1
  Result (C++)     x = 37037000
  Result (ASM)     x = 37037000
Exit? - (y/n)
================ Test #3 ================
      x=(2*a+b*c)/(d-a);   int x,a,b=-333333,c=1000,d=-10;
      Enter a [-32768..32767], a!= -10 ====> 11111
  Result (C++)     x = 29971
  Result (ASM)     x = 29971
Exit? - (y/n)
================ Test #4 ================
      x=(2*a+b*c)/(d-a);   int x,a,b=-333333,c=1000,d=-10;
      Enter a [-32768..32767], a!= -10 ====> -2
  Result (C++)     x = 41666625
  Result (ASM)     x = 41666625
Exit? - (y/n)
================ Test #5 ================
      x=(2*a+b*c)/(d-a);   int x,a,b=-333333,c=1000,d=-10;
      Enter a [-32768..32767], a!= -10 ====> 11
  Result (C++)     x = 15872998
  Result (ASM)     x = 15872998
Exit? - (y/n)
================ Test #6 ================
      x=(2*a+b*c)/(d-a);   int x,a,b=-333333,c=1000,d=-10;
      Enter a [-32768..32767], a!= -10 ====> 9
  Result (C++)     x = 17543841
  Result (ASM)     x = 17543841
Exit? - (y/n)
================ Test #7 ================
      x=(2*a+b*c)/(d-a);   int x,a,b=-333333,c=1000,d=-10;
      Enter a [-32768..32767], a!= -10 ====> -11
  Result (C++)     x = -333333022
  Result (ASM)     x = -333333022
Exit? - (y/n)
```

Тестовые примеры работают нормально.

Попробуйте в этом примере **сами** использовать расширенную команду умножения **IMUL** — см. п. 12.4.1.4.

12.5. Особенности ассемблирования

Сделаем модуль на Ассемблере, иллюстрирующий *новые режимы адресации* (см. п. 12.2.3), назовем его **address.asm,** откомпилируем:

```
tasm32.exe address.asm /l
```

Получим файл листинга. Сравним машинный код с тем, который мы анализировали в п. 8.2.

Файл address.LST

```
Turbo Assembler   Version 5.3 10-07-01    16:57:20      Page 1
address.asm
12.2.3.         Режимы адресации
  1                                       .386
  2  00000000                             model Large,C
  3  00000000                             .CODE
  4  00000000 0021                        d                dw  33
  5  00000002 56CE                        x                dw  22222
  6  00000004 64*(0456)                   brr              dw  100 dup(1110)
  7  000000CC 0153158E                    h                dd  22222222
  8  000000D0 32*(00000007)               htable     dd  50   dup(7)
  9  00000198 0003F28CBDB4DDFD            a       dq   1111111222222333
 10                              ; 1.  Регистровый режим адресации
 11  000001A0 F7 EB                       IMUL   EBX
 12  000001A2 0F AF CB                    IMUL   ECX,EBX
 13  000001A5 8A C7  '                    MOV    AL,BH   ;   8-разрядный
 14  000001A7 66| 8B C3                   MOV    AX,BX ; 16-разрядный
 15  000001AA 8B C3                       MOV    EAX,EBX ; 32-разрядный
 16                              ; 2.  Непосредственный режим адресации.
 17  000001AC  80 C4 06                   ADD    AH,   6
 18  000001AF 66| 05 3039                 ADD    AX, 12345
 19  000001B3 05 0001E240                 ADD    EAX,123456
 20  000001B8 66| 69 C0 04DD              IMUL   AX,1245
 21  000001BD 69 D2      000000DE                IMULEDX,222
 22  000001C3 66| 6B CB 79                IMUL   CX,BX,121
 23  000001C7 69 CB      0012D681         IMUL   ECX, EBX,1234561
 24  000001CD 69 C0      000186A0         IMUL   EAX, 100000
 25  000001D3 66| 2E: 6B 1D          +    IMUL   BX, x,5
 26           00000002r 05
 27  000001DC 66| 2E: 69 1D          +    IMUL   BX, x,50000
 28           00000002rC350
 29  000001E6 2E: 6B 3D 000000CCr +       IMUL   EDI,h,123
 30           7B
 31                              ; 3.  Режим прямой адресации.
 32  000001EE 66| 2E: FF 0D  +            DEC    WORD   PTR h[500]
 33           000002C0r
 34                                    ; аналог предыдущей команды:
 35  000001F6 66| 2E: FF 0D          +    DEC    WORD   PTR h+500
 36           000002C0r
 37  000001FE 2E: FF 05 00000198r         INC    DWORD PTR a
 38                              ; 4.  Режим регистровой косвенной адресации.
 39  00000205 66| 67| 2B 14               SUB    DX, [SI]
 40  00000209 2B 11                       SUB    EDX,[ECX]
 41                              ; 5.  Базовая адресация.
 42  0000020B 66| 67| 03 57 64            ADD    DX, [BX]+100
 43  00000210 2B 50 0A                    SUB    EDX, [EAX]+10
 44                              ; 6.  Индексная адресация.
 45  00000213 66| 2E: 67| 03 96      + ADD    DX,   brr[BP]
 46           0004r
 47  0000021A 2E: 2B 80 000000D0r         SUB    EAX, htable[EAX]
 48                              ; 7.  Индексная адресация с масштабированием
 49  00000221 2E: 2B 04 85      +         SUB    EAX, htable[EAX*4]
 50           000000D0r
 51                              ; 8.  Базово-индексная адресация.
```

```
00000229  03 04 33                    ADD   EAX,  [EBX+ESI]
                ; 9. Базово-индексная адресация с масштабированием.
0000022C  2B 0C C3                    SUB   ECX,  [EBX+EAX*8]
                ; 10. Базово-индексная адресация со  смещением.
0000022F  2B 54 01     0A             SUB   EDX,  ECX[EAX+10]
00000233  03 84 2E     0000ABCF       ADD   EAX,  [ESI][EBP+00ABCFh]
                                    ; аналог предыдущей команды:
0000023A  03 84 2E     0000ABCF       ADD   EAX,  [ESI+EBP+00ABCFh]
00000241  66| 2E: 67| 0F AF 9A+       IMUL  BX,   d[BP+SI]
          0000r
00000249  0F AF 84 2E 0000ABCF        IMUL  EAX,  [ESI][EBP+00ABCFh]
    ; 11. Базово-индексная адресация со смещением и масштабированием.
00000251  2E: 0F AF 8C C3 +           Imul  ECX,htable[EBX+36+EAX*8]
          000000F4r
0000025A 2E: 2B 8C C3     +           SUB   ECX,htable[EBX+36+EAX*8]
          000000F4r
                                    ; аналог предыдущей команды:
00000262  2E: 2B 8C C3    +           SUB   ECX,htable[EAX*8][EBX+36]
          000000F4r
                              end
```

Предварительный анализ файла листинга.

1) Команды, использующие 8- разрядные регистры и адреса, дали тот же машинный код, что и прежде — см. главу 8.

2) Всюду, где употребляются 16-разрядные регистры, появился префикс перекрытия размера операнда 66h, о котором шла речь в п. 12.2.3. Компилятор делает это автоматически, чтобы отличить 16- и 32-разрядные регистры.

3) Всюду, где употребляются 16-разрядные адреса, появился префикс перекрытия размера адреса 67h.

4) В данном конкретном случае команды БЕЗ префиксов используют 32-разрядные расширенные регистры и адреса.

5) Все данные находятся в сегменте кода, поэтому в тех командах, где есть смещение, появляется префикс 2E, как это было и при ассемблировании команд с 16-разрядными смещениями — см. п. 8.2.



ЗАМЕЧАНИЕ.

Процессор **i386** определяет размер операндов выполняемой команды, анализируя бит **D** в дескрипторе сегмента. Если бит **D=0**, то длина всех операндов и исполнительных адресов принимается равной **16 бит**. Это — режим, принятый по умолчанию, для *реального режима работы процессора*. Поэтому все команды, использующие 32-разрядные регистры и адреса, в этом случае получают префикс перекрытия размера операнда **66h** и/или **67h** (см. файл **Shift32.lst**). Если бит **D=1**, то длина всех операндов и исполнительных адресов принимается равной 32 бита. Тогда, наоборот, все команды, использующие 16-разрядные регистры и адреса, получают префикс перекрытия размера операнда **66h** и/или **67h** (как в анализируемом нами файле **address.LST**).

Проанализируем некоторые строки листинга.

```
14   000001A7   66| 8B C3        MOV   AX,BX     ; 16-разрядный
15   000001AA   8B C3            MOV   EAX,EBX   ; 32-разрядный
```

Разница между 16-разрядной и 32-разрядной командами *с регистровой адресацией операндов* только в наличии или отсутствии префикса перекрытия размера операнда **66h** (opSize Prefix). Байт кода операции (КОП) и байт способа адресации (БСА) АБСОЛЮТНО одинаковые. Анализ подобных машинных кодов мы уже делали в п. 8.2.

```
21   000001BD   69 D2      000000DE   IMUL   EDX,222
```

Из табл. П7.1. получаем, что **КОП=69** соответствует расширенной команде **IMUL r/m,im16**. Константа **222** в команде расширилась до 32-битной (**000000DE**). **БСА= D2.**

	Байт кода операции										**Байт способа адресации**							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	0	1	1	0	1	0	0	1			1	1	0	1	0	0	1	0
	КОП						d	w			mod		reg			r/m		

Проанализируем БСА совместно с КОП. Из табл. 8.3 получаем: **mod=11**, значит, **r/m=reg=010=DX** (**w=1**, см. табл. 8.1). Значение собственно поля **reg=010=DX**. Бит **d=0**, значит, регистр **DX** является приемником. Поскольку префикс перекрытия размера операнда отсутствует, то в силу замечания по пункту 4 предварительного анализа файла листинга наш регистр **DX** будет компьютером восприниматься как расширенный, т.е. **EDX**. Как видите, опять те же самые принципы построения машинной команды.

```
23   000001C7   69 CB      0012D681   IMUL   ECX, EBX,1234561
```

Эта команда имеет тот же код операции, что и предыдущая. Разница только в том, что появляется еще один операнд **EBX**, поэтому и изменился байт способа адресации (**БСА= CB**).

	Байт кода операции										**Байт способа адресации**							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	0	1	1	0	1	0	0	1			1	1	0	0	1	0	1	1
	КОП						d	w			mod		reg			r/m		

Проанализируем БСА совместно с КОП. Из табл. 8.3 получаем: **mod=11**, значит, **r/m=reg=011=BX** (**w=1**, см. табл. 8.1). Значение собственно поля **reg=001=CX**. Бит **d=0**, значит, регистр **CX** является приемником. Поскольку префикс перекрытия размера операнда отсутствует, то наши регистры **BX** и **CX** будут компьютером восприниматься как расширенные, т.е. **EBX** и **ECX**.

```
7    000000CC   0153158E       h        dd   22222222
.............................................................
29   000001E6   2E: 6B 3D 000000CCr +IMUL   EDI,h,123
30        7B
```

Эта команда умножения имеет уже другой код операции **КОП=6В (IMUL r/m, im8)**. Но в ней тоже три операнда, и в отличие от предыдущей команды один из операндов находится в памяти (смещение **000000CCr** в сегменте кода, поэтому появляется префикс **2E**). В связи с этим вместо байта способа адресации появился расширенный байт **SIB** – см. п. 12.2.2. **SIB = 3D**.

	Байт кода операции							
Биты	7	6	5	4	3	2	1	0
	0	1	1	0	1	0	1	1
	КОП						d	w

	Байт SIB							
Биты	7	6	5	4	3	2	1	0
	0	0	1	1	1	1	0	1
	S		I			B		

Проанализируем SIB совместно с КОП. Из табл. 12.7 получаем: **S=00**, значит, масштаб при адресации НЕ используется. Из табл. 12.8 следует: **I=111=EDI,** значение поля **B=101** НЕ анализируется, поскольку **S=00**. Бит **d=1**, значит, регистр **EDI** является приемником.

```
39    00000205   66| 67| 2B 14          SUB    DX, [SI]
```

Это обычная команда с 16-разрядными операндами, но, поскольку она используется в 32-разрядном программировании, то в машинном коде появляются два префикса **66h** (opSize Prefix) и **67h** (AddrSize Prefix). **КОП=2B (SUB r16, r/m**– см. табл. П7.1)**, БСА=14**.

	Байт кода операции							
Биты	7	6	5	4	3	2	1	0
	0	0	1	0	1	0	1	1
	КОП						d	w

	Байт способа адресации							
Биты	7	6	5	4	3	2	1	0
	0	0	0	1	0	1	0	0
	mod		reg			R/m		

Проанализируем БСА совместно с КОП. Из табл. 8.3 получаем: **mod=00**, значит, **r/m=100=[SI]**. Значение собственно поля **reg=010=DX (w=1)**. Бит **d=1**, значит, регистр **DX** является приемником.

```
40    00000209   2B 11                  SUB    EDX,[ECX]
```

Эта 32-разрядная команда имеет тот же код операции, что и предыдущая, но в качестве индекса используется другой регистр, НЕ разрешенный в 16-разрядном программировании. В связи с этим вместо байта способа адресации появился расширенный байт **SIB** – см. п. 12.2.2. **SIB =11**.

	Байт кода операции							
Биты	7	6	5	4	3	2	1	0
	0	0	1	0	1	0	1	1
	КОП						d	w

	Байт SIB							
Биты	7	6	5	4	3	2	1	0
	0	0	0	1	0	0	0	1
	S		reg=I			R/m=B		

Проанализируем SIB совместно с КОП. Из табл. 12.7 получаем: **S=00**, значит, масштаб при адресации НЕ используется. **Reg=I=010=EDX** (см. табл. 12.8). Значение поля **R/m=B=001=ECX**. Бит **d=1**, значит, регистр **EDX** является приемником.

```
43    00000210   2B 50    0A            SUB    EDX,[EAX]+10
```

Это — 32-разрядная команда *с адресацией по базе*. Она имеет тот же код операции **2B**. В качестве базы используется другой регистр, НЕ разрешенный в 16-разрядном программировании. В связи с этим вместо байта способа адресации появился байт расширенного способа адресации (**SIB=50**), в котором кодируется информация о приемнике и источнике, а также байт смещения 10 (**0A**).

	Байт кода операции										Байт расширенного способа адресации (SIB)							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	0	0	1	0	1	0	1	1			0	1	0	1	0	0	0	0
	КОП						d	w			S		reg=I			R/m=B		

База **B=000=EAX.** Значение собственно поля **reg=010=EDX**. Бит **d=1**, значит, регистр **EDX** является приемником. Хотя множитель в данной команде есть, но нет индексного регистра, поэтому он никак НЕ учитывается.

```
8 000000D0   32*(00000007)            htable   dd   50  dup(7)
.....................................................................
64 00000251  2E: 0F AF 8C C3 +   Imul CX,htable[EBX+36+EAX*8]
65              000000F4r
```

Это — полноценная 32-разрядная команда. Она имеет расширенный код операции **0F** (Extnsn OpCode — см. табл. П7.1), состоящий из двух байт: **0F AF**, и самый сложный тип адресации — *базово-индексная адресация со смещением и масштабированием*, поэтому в машинном коде присутствуют и байт способа адресации (**БСА=8C**), и его расширение (**SIB=C3**). Смещение **000000F4r** = **000000D0h+24h** (36) находится в сегменте данных, поэтому в машинном коде присутствует префикс **2E**. В байте способа адресации кодируется информация о приемнике и частично, об источнике.

	Байт способа адресации							
Биты	7	6	5	4	3	2	1	0
	1	0	0	0	1	1	0	0
	mod		reg			R/m		

Из табл. 8.3 получаем: **mod=10**, значит, **r/m=100=[SI]+смещение**. Что в действительности является индексом, уточняется в поле **Index** байта **SIB**. Значение собственно поля **reg=001=ECX**.

	Байт SIB							
Биты	7	6	5	4	3	2	1	0
	1	1	0	0	0	0	1	1
	S		I			B		

Проанализируем байт **SIB**. Из табл. 12.7 получаем: **S=11**, значит, масштаб равен 8. **I=000=EAX** (см. табл. 12.8). Этот индекс умножается на масштаб: **EAX*8.** Значение поля базы **B=011=EBX**.



Математический сопроцессор

Полезнее знать несколько мудрых правил, которые могли бы служить тебе, чем выучиться многим вещам, для тебя бесполезным.

Сенека Луций Анней мл.
(ок. 4 г. до н.э. — 65 г. н.э.)

До сих пор мы с вами рассматривали команды обработки целочисленных данных (чисел с фиксированной точкой). Но мы знаем, что есть еще вещественные данные (числа с плавающей точкой). С внутренним форматом таких данных мы с вами познакомились в п. 2.3. Для их обработки служит специальное устройство — *математический сопроцессор* (**FPU — Floating Point Unit**). С момента своего возникновения сопроцессор расширял вычислительные возможности основного процессора **i8086 (i80286, i80386, i80486)** и сначала был выполнен в виде отдельной микросхемы **i8087 (i80287, i80387, i80487)**. Его присутствие в первых моделях процессора было не обязательным. Если сопроцессора не было, то его команды можно было *эмулировать* программным путем, что немного ухудшало производительность основного процессора. Начиная с семейства процессоров **i486DX**, сопроцессор стал составной частью основного процессора (см. табл. 3.1).

Современный сопроцессор обеспечивает полную поддержку **стандартов IEEE-754 и IEEE-854** по представлению и обработке чисел с плавающей точкой. Он может выполнять *трансцендентные операции* (вычисление тригонометрических функций, логарифмов и проч.) с большей точностью.

13.1. Типы данных

Сопроцессор может выполнять операции с семью **обычными** типами данных (см. табл. 13.1), а также со **специальными (особыми) числами** — см. п. 13.1.2.

13.1.1. Обычные данные

Обычные данные могут быть вещественными или целыми числами **со знаком**. Наличие целочисленных данных позволяет выполнять так называемые *смешанные вычисления*, когда в качестве операндов используются и целые, и вещественные данные. Вы можете сравнить данные сопроцессора с теми, которыми оперируют наши базовые алгоритмические языки — см. прил. 4.

Сопроцессор выполняет все вычисления в 80-битном расширенном вещественном формате, а 16-, 32- и 64-битные данные используются для обмена данными и могут применяться в командах сопроцессора как операнды (приемник или источник).

Таблица 13.1. Типы обычных данных сопроцессора.

Тип данных	Длина (бит)	Кол-во значащих цифр	Диапазон представления
Целое слово	16	4-5	-32768 ... 32767
Короткое целое	32	9-10	-2147483648 ... -2147483647
Длинное целое	64	18-19	-9223372036854775808 ... 9223372036854775807
Упакованное двоично-десятичное	80	18	-999999999999999999 ... 999999999999999999
Короткое вещественное	32	7-8	1.175494351e-38 ... 3.402823466e+38
Длинное вещественное	64	15-16	2.2250738585072014e-308 ... 1.7976931348623158e+308
Расширенное вещественное	80	19-20	3.3621031431120935063e-4932 ... 1.189731495357231765e+4932

13.1.2. Особые числа

Кроме обычных чисел, представленных в табл. 13.1, спецификация стандарта IEEE предусматривает несколько специальных форматов, которые могут получиться в результате выполнения математических операций сопроцессора и над которыми можно выполнять отдельные операции (см. п. 13.3).

- **Положительный ноль** — все биты числа сброшены в ноль:
- **Отрицательный ноль** — знаковый бит равен 1, остальные биты числа сброшены в ноль.
- **Положительная бесконечность** — знаковый бит равен 0, все биты экспоненты установлены в 1, а биты мантиссы сброшены в 0.
- **Отрицательная бесконечность** — знаковый бит равен 1, все биты экспоненты установлены в 1, а биты мантиссы сброшены в 0.
- **Денормализованные числа** — все биты экспоненты сброшены в 0. Эти числа позволяют представлять очень маленькие числа при вычислениях с расширенной точностью.

- **Неопределенность** — знаковый бит равен 1, первый бит мантиссы равен 1 (для 80-разрядных чисел первые два бита равны 11), остальные биты мантиссы сброшены в ноль, все биты экспоненты установлены в 1.
- **Не-число типа SNAN (сигнальное)** — первый бит мантиссы равен 0 (для 80-разрядных чисел первые два бита равны 10), остальные биты мантиссы могут содержать 0 и 1, все биты экспоненты установлены в 1.
- **Не-число типа QNAN (тихое)** — первый бит мантиссы равен 1 (для 80-разрядных чисел первые два бита равны 11), остальные биты мантиссы могут содержать 0 и 1, все биты экспоненты установлены в 1.
- **Неподдерживаемое число** — все остальные ситуации.

Пример 13.1 (см. п. 13.4.2) содержит описания и интерпретацию этих чисел в Ассемблере и C++.

13.2. Регистры

Сопроцессор имеет восемь регистров для хранения данных **R0-R7** и пять вспомогательных регистров, которые составляют среду сопроцессора (*Environment*).

Название регистров	Биты															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CWR	Регистр управления (**Control Word Register**)															
SWR	Регистр состояния сопроцессора (**Status Word Register**)															
TWR	Слово признаков-тегов (**Tags Word Register**)															
FIP (32 бита)	Регистр указателя последней выполненной команды (**Floating Instruction Pointer**)															
FDP (32 бита)	Регистр, где хранится адрес операнда последней выполненной команды (**Floating Data Pointer**)															
R0 - ST(2) (80 бит)	Расширенное вещественное или любое другое допустимое данное сопроцессора															
R1 - ST(3) (80 бит)	Расширенное вещественное или любое другое допустимое данное сопроцессора															
.																

R6 - (TOP)	
вершина	
стека ST	Расширенное вещественное или любое другое допустимое данное сопроцессора
или ST(0)	
(80 *бит*)	
R7 - ST(1)	
(80 *бит*)	Расширенное вещественное или любое другое допустимое данное сопроцессора

РИС. 13.1. *Регистры сопроцессора и понятие вершины стека сопроцессора.*

13.2.1. Регистры данных

Регистры данных сопроцессора **R0-R7** имеют длину 80 бит (т.е. пять 16-разрядных слов) и рассматриваются как **круговой стек**, вершина которого (**TOP**) называется **ST** или **ST(0)** и является плавающей. Принцип работы с круговым стеком сопроцессора аналогичен обычному калькулятору. Любая команда загрузки данных сопроцессора автоматически перемещает вершину стека сопроцессора: **TOP=TOP+1**. На рис. 13.1 показана гипотетическая ситуация, когда в результате выполнения какой-то команды вершиной стека стал регистр **R6**. Остальные регистры распределяются подряд по кругу: **R7-ST(1)**, **R0-ST(2),...,R5-ST(7)**. Это и есть их текущие имена **ST(i), i=1,...,7** на момент выполнения данной команды сопроцессора. Если в этих регистрах есть данные, то они могут служить операндами в командах сопроцессора. Обращаться же напрямую к регистрам **R0-R7** нельзя.

13.2.2. Регистр состояния сопроцессора

Этот регистр сигнализирует сопроцессору о его состоянии и о том, как выполнилась та или иная арифметическая команда. Таким образом, этот регистр выполняет функции, аналогичные функциям регистра флагов процессора FLAGS. За битами регистра состояния сопроцессора закреплены соответствующие имена, облегчающие понимание их назначения (для знающих английский язык).

Регистр (слово) состояния сопроцессора (SWR)															
Старший байт								**Младший байт**							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	C3	TOP			C2	C1	C0	ES	SE	PE	UE	OE	ZE	DE	IE
Аналогия с регистром флагов процессора (Flags)															
	ZF				PF		CF								

РИС. 13.2. *Регистр состояния сопроцессора SWR.*



ОБРАТИТЕ ВНИМАНИЕ

У сопроцессора НЕТ аналога знаковому флагу **SF**. Это ОЧЕНЬ существенный момент, который нужно учитывать при организации вычислений с разветвлениями для сопроцессора — см. п. 13.5.3.

Таблица 13.2. Назначение битов регистра состояния сопроцессора (SWR).

Бит	Назначение
Флаги условий (Conditions)	
C0	Аналог флага переноса **CF** (Carry Flag) процессора. Содержит признак переноса информации из старшего бита (после арифметических операций и операций сравнения).
C1	Флаг условия сопроцессора. Прямой аналогии с флагами условий процессора НЕ имеет, он связан с флагом **SE**.
C2	Аналог флага четности **PF** (Parity Flag) процессора.
C3	Аналог флага нуля **ZF** (Zero Flag) процессора. Он устанавливается в 1, если результат выполнения команды равен нулю.
Флаги исключений (Exceptions)	
ES	Общий флаг ошибки (поддерживается, начиная с **i80287**). Устанавливается в 1, если произошла хотя бы одна немаскированная ошибка.
SE	Ошибка стека (поддерживается, начиная с **i80287**). Если **C1=1**, то произошло переполнение стека (команда пыталась записывать информацию в непустую ячейку в стеке). Если **C1=0**, то произошло антипереполнение стека (команда пыталась сосчитать информацию из пустой ячейки в стеке).
PE (Precision)	Флаг неточного результата (потеря точности). Например, результат операции 1/3 не может быть представлен точно.
UE (Underflow)	Флаг антипереполнения – результат слишком мал.
OE (Overflow)	Флаг переполнения – результат слишком велик.
ZE (Zero divide)	Флаг деления на ноль.
DE (Denormal)	Флаг денормализованного операнда – выполнена операция над денормализованным числом.
IE (Invalid operation)	Флаг недопустимой операции.
Прочие флаги состояний	
B (Busy)	Флаг занятости **FPU**. Существует для совместимости с **i8087** и **i80287**. В современных сопроцессорах его значение совпадает со значением флага **ES**.
TOP	Двоичное число от 000 до 111, которое показывает, какой из регистров **R0-R7** в настоящий момент является вершиной стека **ST**.

13.2.3. Регистр управления

Этот регистр определяет для сопроцессора варианты обработки вещественных чисел. На рис.13.3 показаны формат и кодировка полей в управляющем слове (регистре управления). Назначение битов приведено в табл. 13.3.

Регистр (слово) управления сопроцессором (CWR)															
Старший байт								Младший байт							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			IC	RC		PC		IEM		PM	UM	OM	ZM	DM	IM

РИС. 13.3. *Регистр управления сопроцессором CWR.*

. Младшие байты слова управления определяют маскирование (запрет) вычислительных исключений. Биты 0-5 содержат индивидуальные маски для каждого из шести исключений, опознаваемых сопроцессором при выполнении операций с плавающей точкой. Если какой-либо из маскируемых битов установлен (равен 1), то данное исключение запрещается (соответствующий ему флаг в регистре состояния **SWR** НЕ устанавливается — см. табл. 13.2) и процессор продолжит работу, — подробнее см. п. 13.3.

Старшие биты (8-12) управляющего слова определяют параметры работы сопроцессора.

Таким образом, программист может замаскировать особые случаи, задать точность вычислений, способ округления и интерпретацию бесконечности посредством формирования в оперативной памяти нужного ему слова и последующей загрузки его в управляющее слово специальной командой **FLDCW**— см. п. 13.4.7.

13.2.4. Регистр тегов

Этот регистр содержит восемь пар битов, описывающих содержимое (признаки) каждого регистра данных **R0-R7**:

00 — число;

01 — истинный ноль;

10 — особое число (бесконечность, денормализованное число и т.п. — см. п. 13.1.2);

11 — регистр пуст.

Регистр (слово) тегов (TWR)															
Старший байт								Младший байт							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R7		R6		R5		R4		R3		R2		R1		R0	

РИС. 13.4. *Регистр (слово) тегов (TWR).*

Информация регистра тегов может использоваться программами, которые обрабатывают исключительную ситуацию, возникшую при вычислениях в сопроцессоре, без применения комплексного анализа непосредственных значений. Теговую информацию можно прочитать в оперативную память командами **FSTENV/ FNSTENV** или **FSAVE/FNSAVE** — см. п. 13.4.7.

Таблица 13.3. Назначение битов регистра управления сопроцессором (CWR).

Бит	**Назначение**
Биты маскирования (Masking)	
PM (Precision Mask)	Маска неточного результата (потеря точности).
UM(Underflow Mask)	Маска антипереполнения.
OM (Overflow Mask)	Маска переполнения.
ZM (Zero divide Mask)	Маска деления на ноль.
DM (Denormal Mask)	Маска денормализованного операнда.
IM (Invalid operation Mask)	Маска недопустимой операции.
IEM (Interrupt Enable Mask)	Маска разрешения прерываний (**IEM=0**). Если **IEM=1,** то прерывания центрального процессора НЕ будет даже при возникновении индивидуально НЕЗАМАСКИРОВАННОГО случая.
Биты параметров работы сопроцессора	
PC (Precision Control)	Точность результатов арифметических вычислений (команды **FADDx**, **FSUBx**, **FMULx**, **FDIVx**, **FSQRT**) для вещественных чисел (см. п. 2.3.2 – количество разрядов в мантиссе): 00 – 24 бита (32-битные числа); 10 – 53 бита (64-битные числа); 11 – 64 бита (80-битные числа).
RC (Rounding Control)	Управление округлением: 00 – к ближайшему числу; 01 – к отрицательной бесконечности; 10 – к положительной бесконечности; 11 – отбрасывание (усечение к нулю).
IC (Infinity Control)	Управление бесконечностью. Поддерживается для совместимости с **i8087** и **i80287** (**IC=0** – проективная бесконечность: НЕ учитывается знак бесконечности, **IC=1** – аффинная бесконечность: учитывается знак бесконечности). Современные сопроцессоры не учитывают значения этого бита. Для них принята модель аффинной бесконечности и всегда *+бесконечность* > *-бесконечность*.

13.3. Ситуации—исключения

При выполнении команд сопроцессора могут возникать особые ситуации — исключения (Exceptions). Каждое из исключений x устанавливает в 1 соответствующий флаг **xE** в регистре состояния **SWR** и, если данное исключение не замаскировано (бит **xM=0** в регистре управления **CWR**), то вызывается прерывание процессора, которое определенным образом обрабатывается. Если исключение замаскировано, то соответствующий бит **xM=1**, прерывание процессора не происходит, и по умолчанию выполняются следующие действия:

- **PM=1 (неточный результат)** — результат округляется в соответствии с битами **RC** регистра **CWR**;
- **UM=1 (антипереполнение)** — результат слишком мал, он преобразуется в денормализованное число;
- **OM=1 (переполнение)** — результат преобразуется в бесконечность с соответствующим знаком;
- **ZM=1 (деление на ноль)** — результат преобразуется в бесконечность с соответствующим знаком (знак нуля тоже учитывается);
- **DM=1 (денормализованный операнд)** — вычисление продолжается, как обычно;
- **IM=1 (недействительная операция)** — результат определяется, согласно табл. 13.4.

Таблица 13.4. Результаты недействительных операций.

Операция	Результат
Ошибка стека	Неопределенность
Операция с особым числом	Неопределенность
Операция с числом типа **SNAN**	**QNAN**
Сравнение числа с числом типа **xNAN** (**SNAN** или **QNAN**)	Несравнимы. Значение флагов условий **C0=C2=C3=1.**
Сложение бесконечностей одного знака или вычитание бесконечностей разного знака	Неопределенность
Умножение нуля на бесконечность	Неопределенность
Деление бесконечности на бесконечность или деление 0/0	Неопределенность
Команды **FPREM** и **FPREM1**, если делитель равен 0 или делимое равно бесконечности	Неопределенность
Тригонометрическая операция над бесконечностью	Неопределенность
Вычисляется корень квадратный или логарифм из отрицательного числа	Неопределенность
Команда **FBSTP**, если вершина стека **ST** пуста, содержит число типа **xNAN**, бесконечность или превышает 18 десятичных знаков	Неопределенность
Команда **FXCH**, если один из операндов пуст	Неопределенность

13.4. Система команд

Система команд сопроцессора достаточно простая, если знать ключ и немного понимать английский язык. Для их подключения нужно сделать следующее.

- **Воспользоваться одной из директив Ассемблера: .8087, .287 (.286p), .387(.386p), .487 (.486p).** Необходимо иметь в виду, что не все команды сопроцессора, к сожалению, совместимы сверху вниз. Кроме того, директива использования процессора **.x86p** предполагает компиляцию и работу программы в 32-разрядном режиме. Поэтому результаты расчета, выполненные с использованием команд младших моделей сопроцессора, могут отличаться от результатов, полученных на старших моделях.
- **Сделать инициализацию сопроцессора с помощью команды** FINIT (см. п. 13.4.7).
- **При компиляции использовать** ДОПОЛНИТЕЛЬНЫЙ **ключ /r или /e** (Emulated or Real floating-point instructions). Таким образом, теперь вызов компилятора Ассемблера для стыковки с С++ может иметь следующий вид:

```
tasm.exe Name.asm /l /r /ml
```

13.4.1. Условные обозначения для команд базового сопроцессора

Базовый сопроцессор **i8087** имеет 69 базовых команд, которые подчиняются следующим закономерностям:

- Все они **начинаются** на букву F (Floating).
- **Вторая буква** может быть связана с **форматом** обрабатываемых чисел:

I (Integer) — целые числа;

B (Binary-coded decimal) — упакованное двоично-десятичное число (**BCD**).

Для вещественных чисел специальная буква НЕ выделяется.

- Далее идет **мнемонический код операции (МКОП)**, например:

LD (LoaD) — загрузить данное в вершину стека **ST**;

ST (STore) — выгрузить данное из вершины стека **ST**

и уже известные нам команды **ADD, MUL, DIV** и т.п.

- **Заканчиваться команда** может буквой **R** (Reversed — реверсная операция) и/или **P** (Pop — вытолкнуть результат из стека и освободить вершину стека **ST**).

МКОП	**Обычная операция**	**Реверсная операция**
SUB	Приемник = Приемник – Источник	Приемник = Источник - Приемник
DIV	Приемник = Приемник / Источник	Приемник = Источник / Приемник

РИС. 13.5. *Обычные и реверсные операции.*

Сопроцессоры более поздних моделей, начиная с **i80387**, немного нарушили эту стройную систему условных обозначений, но зато увеличили функциональные возможности команд по обработке данных.

13.4.2. Команды пересылки данных

Все команды этой группы имеют один операнд: либо источник, либо приемник. Загрузка *или извлечение информации происходит в/из вершины стека* **ST(0)**. Обычно вершину стека обозначают через ST, и в командах этой группы она явно в операндах НЕ присутствует.

Таблица 13.5. Команды передачи данных.

Команда	Тип данных	Характеристика операнда	Пример
Команды загрузки данных в стек			
FLD источник	Вещественное	Переменная в ОЗУ или ST(i), i=0,...,7.	FLD a FLD ST(5)
FBLD источник	Типа **BCD**	Переменная в ОЗУ	FBLD aBc
FILD источник	Целое	Переменная в ОЗУ	FILD ka
Команды копирования данных из стека			
FST приемник	Вещественное	Переменная в ОЗУ или ST(i), i=1,...,7.	FST ap FST ST(1)
FIST приемник	Целое	Переменная в ОЗУ	FIST kab
Команды извлечения данных из стека с освобождением вершины стека			
FSTP приемник	Вещественное	Переменная в ОЗУ или ST(i), i=1,...,7.	FSTP app FSTP ST(5)
FBSTP приемник	Типа **BCD**	Переменная в ОЗУ	FBSTP ppp
FISTP приемник	Целое	Переменная в ОЗУ	FISTP kabP
Команда обмена содержимого источника с ST(0)			
FXCH [источник]	Если источник НЕ указан, то считается, что он соответствует **ST(1)**.	ST(i), i=1,...,7.	FXCH FXCH ST(4)

ПРИМЕР 13.1.

Создать в Ассемблере особые числа (см. п. 13.1.2) и выдать их значения в программе на алгоритмическом языке C++.

Решение. Особые числа запишем в 16-ричном виде, вспомнив внутренний формат 64- и 80-битных вещественных чисел (см. п. 2.3.2.2 и п. 2.3.2.3). Пусть эти числа будут находиться в сегменте кода, а внешние переменные расположим в сегменте данных.

Для пересылки особых чисел в C++ воспользуемся командами **FLD** и **FSTP**. Команда **FSTP** каждый раз будет освобождать вершину стека **ST(0)**, т.е. все наши особые числа будут загружаться командой **FLD** *в один и тот же регистр сопроцессора* **R0**.

Если воспользоваться командой **FST**, то тогда данные будут командой **FLD** загружаться последовательно в регистры **R0-R7**, реализуя круговой стек. Для данного примера это НЕ принципиально. Но зачем использовать круговой стек, если он нам не нужен? Т.е. нам НЕ нужно содержимое **ST(1)**, **ST(2)**,...

Исходный текст программы PrimC.cpp

```
/* PrimC.cpp              Borland C++ 5.02
          Проект !!!!!! DOS (standard) !!!!!! asm+cpp файлы
          (c) Copyright 2001 by Голубь Н.Г.
     Лекционный пример: ОСОБЫЕ данные и случаи сопроцессора
*/
#include <iostream.h>
#include <conio.h>
#include "Convert.h"
const char* TITLE = TXT("\n=== ОСОБЫЕ данные сопроцессора ======\n");
const char* ZeroPlus = TXT(" положительный ноль ");
const char* ZeroMinus = TXT(" отрицательный ноль ");
const char* SNAN =
          TXT(" НЕ число типа SNAN (сигнальное — операция НЕ действительна)
const char* QNAN = TXT(" НЕ число типа QNAN (тихое) ");
const char* SMALLasm =
          TXT(" денормализованное число (очень МАЛЕНЬКОЕ число — double) ");
const char* UNCERTAINTY1 = TXT(" НЕопределенность (double) ");
const char* InfinityP = TXT(" положительная бесконечность  ");
const char* InfinityN = TXT(" отрицательная бесконечность  ");
const char* AnyType1 = TXT(" все что угодно 1 ");
const char* AnyType2 = TXT(" все что угодно 2 ");
const char* UNCERTAINTY2 = TXT(" НЕопределенность (long double) ");
// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
double ZeroPlusC, ZeroMinusC, SNANc, QNANc, SMALLasmC,
          UNCERTAINTY1c, InfinityPc, InfinityNc;
long double AnyType1C, AnyType2C, UNCERTAINTY2c;
extern "C"
{
   void show (void);
   void primASM(void);
}
// показать состояние ОСОБЫХ данных сопроцессора
void show()
{
  cout << ZeroPlus << ZeroPlusC << endl;
  cout << ZeroMinus << ZeroMinusC << endl;
  cout << SNAN << SNANc << endl;
  cout << QNAN << QNANc << endl;
  cout << SMALLasm << SMALLasmC << endl;
  cout << UNCERTAINTY1 << UNCERTAINTY1c << endl;
  cout << InfinityP << InfinityPc << endl;
  cout << InfinityN << InfinityNc << endl;
  cout << AnyType1 << AnyType1C << endl;
  cout << AnyType2 << AnyType2C << endl;
  cout << UNCERTAINTY2 << UNCERTAINTY2c << endl;
}
void main(void)
{
   cout << TITLE;
   primASM();
   getch();
}
```

Исходный текст модуля FPU_DIGT.asm

```
       title FPU IEEE-754, IEEE-854
           .8087  ; ПОДКЛЮЧЕНИЕ СОПРОЦЕССОРА
;------------------------------
; 13.1. Пример программы на языке Ассемблера
; для проверки внутреннего представления
; ОСОБЫХ данных сопроцессора
;           tasm FPU_DIGT.asm /l /r /ml
;------------------------------
           .model  large,C
           .DATA
;======== Глобальные переменные C++ ============
           EXTRN  C  ZeroPlusC:QWord
           EXTRN  C  ZeroMinusC:QWord
           EXTRN  C  InfinityPc:QWord
           EXTRN  C  InfinityNc:QWord
           EXTRN  C  SMALLasmC:QWord
           EXTRN  C  UNCERTAINTY1c:QWord
           EXTRN  C  SNANc:QWord
           EXTRN  C  QNANc:QWord
           EXTRN  C  UNCERTAINTY2c:TByte
           EXTRN  C  AnyType1C:TByte
           EXTRN  C  AnyType2C:TByte
           .CODE
;======== ОСОБЫЕ числа сопроцессора i8087 ========
;----- double ----------
; положительный ноль
           ZeroPlus  DQ 0
                     DQ 0.0   ; альтернативная запись
; отрицательный ноль
           ZeroMinus DQ 8000000000000000h
                     DQ -0.0   ; альтернативная запись
; положительная бесконечность +INF
           InfinityP DQ 7FF0000000000000h
; отрицательная бесконечность -INF
           InfinityN DQ 0FFF0000000000000h
; денормализованное число (очень МАЛЕНЬКОЕ число)
           SMALLasm  DQ 000FFFFFFFFFFFFFh
; НЕопределенность
           UNCERTAINTY1  DQ  0FFF8000000000000h
; НЕ число типа SNAN (сигнальное)
           SNAN       DQ 7FF0F0F000000000h
; НЕ число типа QNAN (тихое)
           QNAN       DQ 7FF8000000000000h
;----- extended (long double) ----------
; НЕопределенность
           UNCERTAINTY2  dt  0FFFFC000000000000000h
; все что угодно
           AnyType1    dt  0FFFF8000000000000000h
           AnyType2    dt  0FFFFFFFFFFFFFFFFF000h

           EXTRN C show:FAR ; Функция языка C
           PUBLIC C primASM
```

```
primASM    PROC   large
; ——— Инициализация сопроцессора
           FINIT
;——— аналог ZeroPlusC=ZeroPlus
           FLD   ZeroPlus    ;   ST(0) <===== ZeroPlus
;   ZeroPlusC <===== ST(0) с освобождением вершины стека
           FSTP ZeroPlusC
;————————————————————
           FLD   ZeroMinus   ;   ST(0) <===== ZeroMinus
           FSTP ZeroMinusC
           FLD   InfinityP
           FSTP InfinityPc
           FLD   InfinityN
           FSTP InfinityNc
           FLD   SMALLasm
           FSTP SMALLasmC
           FLD   UNCERTAINTY1
           FSTP UNCERTAINTY1c
           FLD   UNCERTAINTY2
           FSTP UNCERTAINTY2c
           FLD   SNAN
           FSTP SNANc
           FLD   QNAN
           FSTP QNANc
           FLD   AnyType1
           FSTP AnyType1C
           FLD   AnyType2
           FSTP AnyType2C
; показать в C++ состояние ОСОБЫХ данных сопроцессора
           call show
           ret
primASM ENDP
           END
```

Результат работы программы

```
====== ОСОБЫЕ данные сопроцессора =========
   положительный ноль  0
   отрицательный ноль  0
   НЕ число типа SNAN (сигнальное – операция НЕ действительна) +NAN
   НЕ число типа QNAN (тихое) +NAN
   денормализованное число (очень МАЛЕНЬКОЕ число – double)
   2.22507e-308
   НЕопределенность (double) -NAN
   положительная бесконечность   +INF
   отрицательная бесконечность   -INF
   все что угодно 1 -INF
   все что угодно 2 -NAN
   НЕопределенность (long double) -NAN
```

Теперь, если в результате своих вычислений с вещественными числами вы получите вместо ожидаемого числа, например, **-NAN**, вы будете знать, что это такое. Можете сделать аналогичный пример, используя алгоритмический язык Паскаль, и посмотреть результаты.

13.4.3. Команды загрузки констант

Команды этой группы помещают в вершину стека **ST(0)** часто используемые константы. Начиная с сопроцессора **i80387**, *эти константы хранятся в более точном формате.* Команды операндов не имеют.

Таблица 13.6. Команды загрузки констант.

Команда	Назначение
FLD1	Поместить в ST(0) число 1.0
FLDZ	Поместить в ST(0) число +0.0
FLDPI	Поместить в ST(0) число p
FLDL2E	Поместить в ST(0) число, равное $\log_2 e$
FLDL2T	Поместить в ST(0) число, равное $\log_2 10$
FLDLN2	Поместить в ST(0) число, равное ln 2
FLDLG2	Поместить в ST(0) число, равное lg 2

13.4.4. Арифметические команды

Арифметические команды реализуют базовую арифметику сопроцессора (четыре основные арифметические операции: +, -, *, /) и несколько специальных арифметических команд.

Формат команд базовой арифметики **для вещественных чисел** достаточно сложный (операнды могут быть заданы в явном виде, а могут и отсутствовать, т.е. быть неявными), поэтому во избежание недоразумений в табл. 13.7 дан явный формат. К допустимым форматам для этих команд мы еще вернемся в п. 13.5.2.

Таблица 13.7. Команды базовой арифметики.

Команда	Тип данных	Алгоритм выполнения
Команды сложения		
FADD приемник, источник	Вещественное	Приемник = приемник + источник
FADDP приемник, источник	Вещественное	
FIADD источник	Целое	
Команды обычного вычитания		
FSUB приемник, источник	Вещественное	Приемник = приемник - источник
FSUBP приемник, источник	Вещественное	
FISUB источник	Целое	
Команды реверсного (обратного) вычитания		
FSUBR приемник, источник	Вещественное	Приемник = источник – приемник
FSUBRP приемник, источник	Вещественное	
FISUBR источник	Целое	
Команды умножения		
FMUL приемник, источник	Вещественное	Приемник = приемник * источник
FMULP приемник, источник	Вещественное	
FIMUL источник	Целое	

Команды обычного деления		
FDIV приемник, источник	Вещественное	Приемник = приемник/источник
FDIVP приемник, источник	Вещественное	
FIDIV источник	Целое	
Команды реверсного (обратного) деления		
FDIVR приемник, источник	Вещественное	Приемник = источник/приемник
FDIVRP приемник, источник	Вещественное	
FIDIVR источник	Целое	

Специальные арифметические команды в своем формате НЕ содержат операндов, т.е. ВСЕ их операнды заданы НЕЯВНО.

Таблица 13.8. Специальные арифметические команды.

Команда	Назначение	Алгоритм выполнения
FPREM	Нахождение частичного остатка от деления путем последовательного вычитания 64 раза содержимого ST(1) из ST(0)	ST(0)=остаток[ST(0)/ST(1)] Формируется флаг C2 регистра управления **CWR:** C2=0 – получен точный остаток от деления, т.е. остаток<ST(1). C2=1 – точный остаток НЕ получен (получен частичный остаток)
FPREM1 (i80387)	Нахождение частичного остатка от деления в стандарте IEEE – округление к ближайшему целому	
FABS	Нахождение абсолютного значения	ST(0) = abs(ST(0))
FSQRT	Нахождение корня квадратного	ST(0) = sqrt(ST(0))
FCHS	Изменение знака	ST(0) = -ST(0)
FRNDINT	Округление до целого	Содержимое ST(0) округляется до целого в соответствии с битами **RC** регистра управления сопроцессором **CWR**
FXTRACT	Извлечь экспоненту и мантиссу числа	Число ===> ST(0); ST(0) = мантисса числа; ST(1) = экспонента числа
FSCALE	Команда, обратная FXTRACT	ST(0) <=== мантисса числа; ST(1) <=== экспонента числа; $ST(0)=ST(0)*\ 2^{ST(1)}$

13.4.5. Трансцендентные операции

Команды этой группы тоже в своем формате НЕ содержат операндов, т.е. их операнды заданы НЕЯВНО. Они предназначены для вычисления наиболее употребительных арифметических функций.

Таблица 13.9. Команды трансцендентных операций.

Команда	Назначение	Алгоритм выполнения
FSIN (i80387)	Вычисление синуса sin(**x**), где значение аргумента **x** задается в радианах	**x** ===> ST(0); ST(0) = sin(ST(0)) -2^{63}<= **x** <= 2^{63}. Если значение **x** выходит за эти пределы, можно воспользоваться командой FPREM с делителем 2π. Иначе флаг C2=1 и значение ST(0) не изменяется.
FCOS (i80387)	Вычисление косинуса cos(**x**), где значение аргумента **x** задается в радианах	**x** ===> ST(0); ST(0) = cos(ST(0)) -2^{63}<= **x** <= 2^{63}. Если значение **x** выходит за эти пределы, можно воспользоваться командой FPREM с делителем 2 π. Иначе флаг C2=1 и значение ST(0) не изменяется.
FSINCOS (i80387)	Вычисление синуса и косинуса	**x** ===> ST(0); ST(1) = sin(ST(0)); ST(0) = cos(ST(0)) -2^{63}<= **x** <= 2^{63}. Если значение **x** выходит за эти пределы, можно воспользоваться командой FPREM с делителем 2 π. Иначе флаг C2=1 и значение ST(0) не изменяется.
FPTAN[1)]	Вычисление частичного тангенса tg(**a**)=**y**/**x**, где значение аргумента **a** задается в радианах	**a** ===> ST(0); ST(0) = **x**; ST(1) = **y** 0 < **a** < π/4 (i8087, i80287). -2^{63}<= **a** <= 2^{63} (начиная с i80387). Если значение **a** выходит за эти пределы, можно воспользоваться командой FPREM с делителем π. Иначе флаг C2=1 и значение ST(0) не изменяется.
FPATAN	Вычисление арктангенса **a** = arctg (**y**/**x**)	ST(0) = **x**; ST(1) = **y**; **a** <=== ST(0), знак совпадает со знаком ST(1). 0 < |**y**| < |**x**| < бесконечность. |**a**| < π
F2XM1	Вычисление 2^{X}-1	**X** ===> ST(0); ST(0) = 2^{X}-1. -1< **X** <1, иначе результат операции не определен.
FYL2X[2)]	Вычисление **y***$\log_2$(**x**)	**x** ===> ST(0); **y** ===> ST(1); 0 < **x** < бесконечность, -бесконечность < **y** < +бесконечность.
FYL2XP1[3)]	Вычисление **y***$\log_2$(**c**+1)	**c** ===> ST(0); **y** ===> ST(1); 0 <= |**c**| < 1-$2^{0.5}$/2, -бесконечность < **y** < +бесконечность.

 **ЗАМЕЧАНИЯ.**

[1] — это самая запутанная и непонятная для начинающих программистов операция сопроцессора. Как с ней справиться, — см. пример 13.2 (п. 13.5.2).

[2] Если воспользоваться известной в математике формулой:

$$\log_n x = \log_n 2 * \log_2 x,$$

то можно с помощью данной команды и соответствующих констант (см. табл. 13.6) вычислить логарифмы с разными основаниями.

[3] Эта команда применяется для более точного, чем предыдущая команда, вычисления логарифмов для чисел, которые ОЧЕНЬ близки к единице. Известно, что логарифм по любому основанию от единицы равен нулю. Вся сложность при этих вычислениях связана с тем, с какой точностью к этому нулю можно приблизиться.

13.4.6. Команды сравнения

Команды этой группы выполняют сравнение содержимого регистра **ST(0)** с *источником* и выставляют соответствующие флаги сопроцессора или процессора (для команд сравнения сопроцессора, начиная с **Pentium Pro**).

При сравнении **вещественных данных** *источник* может находиться или в регистре ST(i), i=1,...,7, или в оперативной памяти. Если *источник* в формате команды НЕ указан, предполагается, что он хранится в **ST(1).**

При сравнении **целочисленных данных** *источник* может находиться только в оперативной памяти.

Таблица 13.10. Основные команды сравнения.

Команда	**Назначение**
FCOM источник	Сравнение вещественных данных
FCOMP источник	Сравнение вещественных данных и освобождение вершины стека ST(0)
FCOMPP	Сравнение вещественных данных и освобождение вершины стека ST(0) и регистра ST(1)
FUCOM источник (i80387)	Сравнение вещественных данных БЕЗ учета порядков
FUCOMP источник (i80387)	Сравнение вещественных данных БЕЗ учета порядков и освобождение вершины стека ST(0)
FUCOMPP (i80387)	Сравнение вещественных данных БЕЗ учета порядков и освобождение вершины стека ST(0) и регистра ST(1)
FICOM источник	Сравнение целочисленных данных
FICOMP источник	Сравнение целочисленных данных и освобождение вершины стека ST(0)
FCOMI ST(0), ST(i) (Pentium Pro)	Сравнение данных и установка EFLAGS – см. табл. 12.12
FCOMIP ST(0), ST(i) (Pentium Pro)	Сравнение данных, установка EFLAGS и освобождение вершины стека ST(0) – см. табл. 12.12
FUCOMI ST(0), ST(i) (Pentium Pro)	Сравнение данных БЕЗ учета порядков и установка EFLAGS – см. табл. 12.12

FUCOMIP ST(0), ST(i) (Pentium Pro)	Сравнение данных БЕЗ учета порядков, установка EFLAGS и освобождение вершины стека ST(0) – см. табл. 12.12
FTST	Сравнение содержимого ST(0) с нулем
FXAM	Анализ содержимого и определение типа числа в ST(0) – см. табл. 13.13

Таблица 13.11. Флаги сравнения для команд базового сопроцессора.

Условие	Флаг C3	Флаг C2	Флаг C0
ST(0) > источник	0	0	0
ST(0) < источник	0	0	1
ST(0) = источник	1	0	0
Операнды несравнимы	1	1	1

Таблица 13.12. Флаги сравнения для команд сопроцессора Pentium Pro.

Условие	Флаг ZF	Флаг PF	Флаг CF
ST(0) > источник	0	0	0
ST(0) < источник	0	0	1
ST(0) = источник	1	0	0
Операнды несравнимы	1	1	1

Благодаря тому, что сразу формируются флаги **EFLAGS** процессора, команды сравнения для сопроцессора **Pentium Pro** работают быстрее и не изменяют содержимого регистра **AX,** но НЕ все сопроцессоры сторонних от **Intel** производителей (и компиляторы) их поддерживают — см. п. 12.5.3.

Таблица 13.13. Результаты действия команды FXAM.

Тип числа в ST(0)	Флаг C3	Флаг C2	Флаг C0
Не поддерживаемое	0	0	0
НЕ-число	0	0	1
Нормальное конечное число	0	1	0
Бесконечность	0	1	1
Нуль	1	0	0
Регистр пуст	1	0	1
Денормализованное число	1	1	0

Флаг C1 **устанавливается равным знаку числа в** ST(0).

13.4.7. Команды управления сопроцессором

В командах этой группы неявных операндов нет. Приемник или источник соответствуют участку в оперативной памяти необходимой длины.

Таблица 13.14. Основные команды управления сопроцессором.

Команда	Назначение
FLDCW источник	Загрузка регистра управления CWR из источника (слово в оперативной памяти)
FSTCW приемник	Запись содержимого слова управления CWR в приемник (оперативная память). Данная команда полностью эквивалентна команде WAIT FNSTCW
FNSTCW приемник	Запись содержимого слова управления CWR в приемник (оперативная память) без ожидания окончания обработки исключительных ситуаций
FSTSW приемник (i80287)	Запись содержимого слова состояния SWR в приемник (оперативная память). Данная команда полностью эквивалентна команде WAIT FNSTSW
FNSTSW приемник (i80287)	Запись содержимого слова состояния SWR в приемник (оперативная память) без ожидания окончания обработки исключительных ситуаций
FSAVE приемник	Сохранение полного состояния FPU (регистров данных и вспомогательных регистров) в приемнике (участок памяти размером 94 или 108 байт). Данная команда полностью эквивалентна команде WAIT FNSAVE
FNSAVE приемник	Сохранение полного состояния FPU (регистров данных и вспомогательных регистров) без ожидания окончания обработки исключительных ситуаций
FXSAVE приемник (Pentium II)	Быстрое сохранение полного состояния FPU (регистров данных и вспомогательных регистров) в приемнике (участок памяти размером 512 байт). Данная команда НЕ совместима с командами FSAVE/FRSTOR
FRSTOR источник	Восстановление полного состояния FPU (регистров данных и вспомогательных регистров). Данная команда является обратной команде FSAVE
FXSTOR источник (Pentium II)	Быстрое восстановление полного состояния FPU (регистров данных и вспомогательных регистров). Данная команда является обратной команде FXSAVE
FLDENV источник	Загрузка из источника (14 или 28 байт в памяти в зависимости от разрядности операндов) пяти вспомогательных регистров окружения сопроцессора (CWR, SWR, TWR, FIP, FDP)
FSTENV приемник	Сохранение пяти вспомогательных регистров. Данная команда полностью эквивалентна команде WAIT FNSTENV и является обратной команде FLDENV
FNSTENV приемник	Сохранение пяти вспомогательных регистров без ожидания окончания обработки исключительных ситуаций. Данная команда является обратной команде FLDENV
FWAIT WAIT	Ожидание готовности сопроцессора
FINIT	Инициализация сопроцессора. Данная команда полностью эквивалентна команде WAIT FNINIT
FNINIT	Инициализация сопроцессора без ожидания окончания обработки исключительных ситуаций
FENI (только в i8087)	Включение исключений. В старших версиях сопроцессора воспринимается как команда FNOP

FENI (только в i8087)	Включение исключений. В старших версиях сопроцессора воспринимается как команда FNOP
FDISI (только в i8087)	Запрещение исключений. В старших версиях сопроцессора воспринимается как команда FNOP
FNOP	Отсутствие операции
FCLEX	Обнуление флагов исключений в регистре состояния SWR (PE, UE, OE, ZE, DE, IE, ES, SE и B). Данная команда полностью эквивалентна команде WAIT FNCLEX
FNCLEX	Обнуление флагов исключений без ожидания
FINCSTP	Поле TOP регистра состояния сопроцессора увеличивается на 1.
FDECSTP	Поле TOP регистра состояния сопроцессора уменьшается на 1.
FFREE ST(i)	Освобождение регистра данных ST(i), i=0,…,7. В регистре тегов TWR данный регистр помечается как пустой

13.5. Основные особенности программирования

13.5.1. Допустимые операнды

Перечислим основные особенности программирования сопроцессора, связанные с операндами.

1. При анонимных обращениях к памяти желательно задавать тип операнда через директиву **PTR**. Например:

```
53    009E   9B DD 07          FLD   [BX]
*Warning* FPU_DIGT.asm(52) Argument needs type override
                              ;А вот так нормально!
54    00A1   9B DD 07          FLD   QWORD PTR[BX]
```

2. Константы в операндах НЕ допускаются. Их нужно задавать так, как мы это делали в примерах 13.1 или 2.10.

3. Наличие или отсутствие операнда связано не только с самой командой, как это было с большинством команд базового процессора, но и с допустимыми форматами команд сопроцессора. Например, для арифметических команд возможны несколько форматов.

13.5.2. Форматы основных арифметических команд

Арифметические команды сопроцессора (op={**ADD,SUB,MUL,DIV,SUBR,DIVR**}), представленные в табл. 13.7, могут иметь пять форматов (см. табл. 13.15).

Познакомимся с форматами и арифметическими командами на практике. Разберемся для начала с самой загадочной операцией сопроцессора **FPTAN**.

ПРИМЕР 13.2.

Вычислить **y = (tg (a) + b)/(c*c-1)**, где переменные **a, b, c, y** — 32-разрядные вещественные числа.

Решение. Покажем решение нашей задачи в связке Паскаль+Ассемблер. Обратите внимание на то, как в Паскале подключается сопроцессор. Для этого нужны ДВЕ директивы **{$N+}** и **{$E+}**. Функцию вычисления тангенса в Паскале реализуем через известное тригонометрическое соотношение: **tg (a) = sin(a)/cos(a), cos(a)<>0**. Таким образом, мы исключим бесконечность.

В Ассемблере воспользуемся набором команд сопроцессора **i387**, применим команду **FPREM** для приведения аргумента **a** к нужному диапазону. Из математики известно, что тангенс — функция периодическая с периодом π, поэтому делителем для этой команды и будет число π.

Таблица 13.15. Форматы команд базовой арифметики.

Формат	Мнемокод	Предполагаемые операнды (приемник, источник)	Пример
Стековый	F_{op}	ST(1),ST(0)	FSUB
Регистровый	F_{op}	ST(i), ST(0) ST(0), ST(i), i=0,...,7.	FMUL ST(5),ST FADD ST,ST(2)
Регистровый с извлечением из стека	$F_{op}P$	ST(i), ST(0), i=0,...,7.	FMULP ST(3),ST(0)
Вещественные данные в памяти	F_{op}	ST(0), mem_{32} ST(0), mem_{64}	FADD a
Целые данные в памяти	FI_{op}	ST(0), mem_{16} ST(0), mem_{32} ST(0), mem_{64}	FIDIV k

Исходный текст модуля lect8087.asm

```
        title lect8087  (CopyRight by Голубь Н.Г., 1993, 1998, 2001)
        ; 13.2. Пример программы на языке Ассемблера
        ;           y = (tg (a) + b)/(c*c-1)
        ;          tasm LECT8087.ASM /l /r
        ;-------------------------------------------------------------
                  .387
    data    segment
        EXTRN  a:Dword,b:Dword,c:Dword,y:Dword
    data          ends
    code          segment
                  assume cs:code,ds:data
                  public  lecture
    ;------- КОНСТАНТЫ
    one     dd      1.

    lecture proc near
                  finit                     ;инициализация i80387
                                ;-------ST(0)--------!-------ST(1)-------!
                  FLDPI         ;PI                  |?
                  fld     a     ;a                   !PI
                  FPREM         ;остаток a/PI
                  FPTAN         ;X                   !Y
```

```
; СТЕКОВЫЙ формат
      Fdiv                      ;tg(a)=Y/X
; один из операндов (источник) в памяти
      fadd      b               ;tg(a) + b
      fld       c               ;c                  !tg(a) + b
      fmul      c               ;c*c                !tg(a) + b
      fsub      one             ;c*c - 1            !tg(a) + b
; РЕГИСТРОВЫЙ с извлечением из стека формат
      fdivP     st(1),st(0)     ;(tg(a) + b)/(c*c - 1)
; аналогичен СТЕКОВОМУ формату:   FDIV

      fstp      y
                ret
lecture         endp
code            ends
                end
```

В нашем модуле на Ассемблере задействованы только два регистра данных сопроцессора **ST(0)** и **ST(1)**. В примере видно, как они используются.

Исходный текст программы lect8087.pas

```
{ CopyRight by Голубь Н.Г., 1993, 1998, 2001
  Вычислить:   y = (tg (a) + b)/(c*c-1)
   Корректность вводимых числовых символов
  и допустимый диапазон чисел НЕ проверяется!!!
 }
Program lect8087;
{$N+}
{$E+}
{$L lect8087}
Uses CRT;
Label 1,2;
Var a,b,c,y         : Single;
    ch              : char;
Procedure lecture; external;
Procedure Variant;
Begin
    y := (sin (a)/cos (a) + b)/(c*c-1)
End;
Begin
    ch:='y';
    ClrScr;
    writeln ('Вычисляем:');
    writeln ('  y = (tg (a) + b)/(c*c-1) ');
    writeln('  ТИП ДАННЫХ:');
  writeln('  a,b,c : Single');
  while (ch='y') or (ch='Y') do
   begin
      Window(1,6,79,20);
      ClrScr;
      write ('введите a,b,c:  ');
      readln (a,b,c);
      1:if abs(c) =1 then
```

```
    begin
            write ('Повторите ввод значения abs(c) <> 1:   ');
            readln (c);
      goto 1
    end;
    2:if cos (a)=0 then
        begin
            write ('Повторите ввод значения a, т.к. cos (a)<>0):   ');
            readln (a);
            goto 2
        end;
    lecture;
    writeln ('АССЕМБЛЕР: y=',y);
    Variant;
    writeln ('ПАСКАЛЬ:   y=',y);
    writeln ('продолжать? (y/n)');
    ch:=ReadKey;
  end
End.
```

Тестовые примеры:

```
Вычисляем:
   y = (tg (a) + b)/(c*c-1)
   ТИП ДАННЫХ:
   a,b,c : Single
введите a,b,c:   623745.24e11
-2222.22e-33
1
Повторите ввод значения abs(c) <> 1:    1.000111
АССЕМБЛЕР:    y=  4.51649609375000E+0003
ПАСКАЛЬ:      y=  4.51649609375000E+0003
продолжать? (y/n)
----------------------
Вычисляем:
   y = (tg (a) + b)/(c*c-1)
   ТИП ДАННЫХ:
   a,b,c : Single
введите a,b,c:   76825768237896e-22
213213231e-21
43234.423
АССЕМБЛЕР:    y=  4.11016906031831E-0018
ПАСКАЛЬ:      y=  4.11016906031831E-0018
продолжать? (y/n)
----------------------
Вычисляем:
   y = (tg (a) + b)/(c*c-1)
   ТИП ДАННЫХ:
   a,b,c : Single
введите a,b,c:   33
-33
3
АССЕМБЛЕР:    y=-1.35391263961792E+0001
ПАСКАЛЬ:      y=-1.35391263961792E+0001
продолжать? (y/n)
----------------------
```

```
Вычисляем:
   y = (tg (a) + b)/(c*c-1)
   ТИП ДАННЫХ:
   a,b,c : Single
введите a,b,c:  0.2
2
3
АССЕМБЛЕР:    y=  2.75338768959045E-0001
ПАСКАЛЬ:      y=  2.75338768959045E-0001
продолжать?  (y/n)
```

Более полный вариант тестовых примеров можно посмотреть в файле **LECT8087.tst**.

ПРИМЕР 13.3.

Вычислить действительные корни квадратного уравнения: **a*x² + b*x + c = 0,** где переменные **a, b, c** — 32-разрядные **целые** числа.

Решение. Покажем решение нашей задачи в связке Borland C++ и Ассемблер. Здесь будут реализованы так называемые **смешанные вычисления**. Условия существования корней (проверка значения входных значений коэффициентов **a** и **b**, а также дискриминанта **d=b²-4*a*c**) пока в Ассемблере делать НЕ будем. Возложим это на С++. С целью проверки вычислений передадим из Ассемблера в С++ вычисленные значения **bb= b²**, **ac=4*a*c** и дискриминанта **d.** Подробно проиллюстрируем с помощью комментариев, что происходит с регистрами данных сопроцессора **ST(0)** и **ST(1).**

Исходный текст модуля Quadr.asm

```
    title real8087  (CopyRight by Голубь Н.Г., 1993, 1998, 2001)
    ; 13.3. Пример программы на языке Ассемблера
    ; Вычислить действительные корни квадратного уравнения:
    ;           a*x*x + b*x + c = 0
    ;          TASM.EXE QUADR.ASM /l /r /ml
    ;  ВСЕ особые ситуации обрабатываются в С++
    ;-----------------------------------------------------------
    .387
    .model large,C
    .data
EXTRN    C     a:Dword,b:Dword,c:Dword,x1:dword,x2:dword,d:dword
EXTRN    C     ac:Dword,bb:Dword
    .code
    public  quadr
;------ КОНСТАНТЫ
four     dd    4.
two      dd    2.
quadr    proc C far
    finit                       ;инициализация 8087
                                ;------ST(0)------!------ST(1)------!
    fild  b                     ;b              !?
    fmul  st(0),st(0)           ;b*b            !?
    FST   bb                    ;копирование вершины стека ==> bb
    fild  a                     ;a              !b*b
    fmul  four                  ;4*a            !b*b
    fimul c                     ;4*a*c          !b*b
    FST   ac       ;копирование вершины стека ==> ac
```

```
;——————————————— альтернативные решения:
    ;fsubrp  st(1),st(0)   .;d=4*a*c-b*b            !?
    ;fsub    st(0),st(1)   ;d=4*a*c-b*b             !b*b
    ;fchs                  ;d=-4*a*c+b*b            !?
;———————————————————————————————————————————————————————
    fsubP    st(1),st(0)   ;d=b*b-4*a*c             !?
    fst      d             ;копирование вершины стека ==> d
    fsqrt                  ;sqrt(d)                 !?
    fld      st(0)         ;sqrt(d)                 !sqrt(d)
    fchs                   ;-sqrt(d)                !sqrt(d)
    fiadd    b             ;b-sqrt(d)               !sqrt(d)
    fchs                   ;-b+sqrt(d)              !sqrt(d)
    fxch     st(1)         ;sqrt(d)                 !-b+sqrt(d)
    fiadd    b             ;b+sqrt(d)               !-b+sqrt(d)
    fchs                   ;-b-sqrt(d)              !-b+sqrt(d)
    fidiv    a             ;-b-sqrt(d)/a            !-b+sqrt(d)
    fdiv     two           ;-b-sqrt(d)/a/2          !-b+sqrt(d)
    fstp     x2            ;-b+sqrt(d)              !?
    fidiv    a             ;-b+sqrt(d)/a            !-b+sqrt(d)
    fdiv     two           ;-b+sqrt(d)/a/2          !?
    fstp     x1            ;?                       !?
    ret
quadr        endp
    end
```

Исходный текст программы Pr13_3.cpp

```
/* Pr13_3.cpp                Borland C++ 5.02
          Проект !!!!!! Application.exe === > DOS Standard !!!!!!
                          asm+cpp файлы
           (c) Copyright 1998-2001 by Голубь Н.Г.
                      Лекционный пример 13.3.
      Вычислить действительные корни квадратного уравнения:
                a*x*x + b*x + c = 0
      Корректный ввод числовых символов.
             Поддержка КИРИЛЛИЦЫ.
*/
#include <iomanip.h>
#include <conio.h>
#include <math.h>
#include "convert.h"  // Перекодировка КИРИЛЛИЦЫ Win—>DOS
#include "inputNum.h" // Ввод ЛЮБЫХ ЦЕЛЫХ чисел

const char* InputA = TXT(" Введите значение a: ");
const char* InputB = TXT(" Введите значение b: ");
const char* InputC = TXT(" Введите значение c: ");
const char* ResultC =   TXT("\n\n Результат в C++: ");
const char* ResultASM = TXT("\n\n Результат в ASM: ");
const char* CONTINUE = TXT("\n\n Нажмите любую клавишу для продолжения..
const char* CALC =
      TXT("\n\tВычислить действительные корни квадратного уравнения:\n");

const char* UNCOUNTABLE = TXT("\nБесчисленное множество решений\n");
const char* LINEAR = TXT("\nУравнение линейное");
const char* NOROOTS = TXT("\n!!!!!!! Нет вещественных корней  !!!!!!\n");
```

```
//———————— ГЛОБАЛЬНЫЕ переменные — передаются в Ассемблер
long int a,b,c;
float x1,x2,d,ac,bb;
//————————————————————————————————————

// ВНЕШНИЕ ассемблерные функции
extern "C"
{
 void quadr (void);
}
 void title(void)
{
 cout << CALC;
 cout << "            a*x*x + b*x + c = 0\n";
}
 void title(long int a,long int b, long int c)
{
 cout <<"\n\t  a = "<< a << ";  b = " << b << ";  c = " << c ;
}
void PressAnyKey()
{ cout << CONTINUE;
  getch();
}

int quadrC(long int a, long int b, long int c);

void cond (long int a, long int b, long int c)
{
  if (!a)//(a==0)
    if (!b) cout << UNCOUNTABLE;
     else
           { cout << LINEAR;
              float x=-(float)c/b;
              cout << "\n    x=" << x << endl;
           }
          else
          if(quadrC(a,b,c))
             cout << NOROOTS;
          else
                {
                   cout << "\n x1=" << setprecision(20)<< x1 << "   x2=" << x2 << endl;
                   x1=x2=0;
                   cout << ResultASM;
                   quadr();
                   cout    << "\n x1=" << x1 << "   x2=" << x2 << endl;
                   cout    << "\n     d=" << setprecision(20)<< d << "\n   b*b=" << bb
                           << ";   4.*a*c=" << ac << endl;
      }
}
int quadrC(long int a, long int b, long int c)

{ bb=(float)b*b;
   ac=4.*a*c;
   d=bb-ac;
   cout    << ResultC;
   cout    << "\n     d=" << setprecision(20)<< d << "\n   b*b=" << bb
           << ";   4.*a*c=" << ac << endl;
```

```
      if (d<0)return 1;
      else
       { x1=(-b+sqrt(d))/(2.*a);
         x2=(-b-sqrt(d))/(2.*a);
        return 0;
        }
}

   void main()
   { int t=0;
     for(;;) // бесконечный цикл
       {
          clrscr();
          cout << "———————————— Test #" << ++t << " ————————————";
          title();
          while (input (InputA,a));
           while (input (InputB,b));
           while (input (InputC,c));
          title (a,b,c);
          cond(a,b,c);
          PressAnyKey();
       }
}
```

```
 ———————————— Test #1 ————————————
        Вычислить действительные корни квадратного уравнения:
              a*x*x + b*x + c = 0
   Введите значение a: (long) === >7867867869
   Ошибка! НЕ корректен диапазон для типа long
   Введите значение a: (long) === > 897897
   Введите значение b: (long) === > -7687689
   Введите значение c: (long) === > -675
              a = 897897;  b = -7687689;  c = -675
   Результат в С++:
        d=59102986240000
        b*b=59100561932288;        4.*a*c=-2424321792
        x1=8.5619707107543945З     x2=-8.7793028797022998Зe-05
   Результат в ASM:
        x1=8.5619707107543945З     x2=-8.7801818153820931 9e-05
        d=59102986240000
        b*b=59100561932288;        4.*a*c=-2424321792
   Нажмите любую клавишу для продолжения...
 ———————————— Test #2 ————————————
   Вычислить действительные корни квадратного уравнения:
              a*x*x + b*x + c = 0
   Введите значение a: (long) === > -6875
   Введите значение b: (long) === > 786
   Введите значение c: (long) === > 8977898970
   Ошибка! НЕ корректен диапазон для типа long
   Введите значение c: (long) === >6785786
              a = -6875;  b = 786;  c = 6785786
   Результат в С++:
        d=186609729536
        b*b=617796;                4.*a*c=-186609106944
        x1=-31.3598175048828125    x2=31.4741439819335938
   Результат в ASM:
        x1=-31.3598175048828125    x2=31.4741439819335938
        d=186609729536
        b*b=617796;                4.*a*c=-186609106944
```

```
Нажмите любую клавишу для продолжения...
 ———————————— Test #3 ————————————
              Вычислить действительные корни квадратного уравнения:
                a*x*x + b*x + c = 0
  Введите значение a: (long) === > 0
  Введите значение b: (long) === > 0
  Введите значение c: (long) === > 1
                a = 0;  b = 0;  c = 1
 Бесчисленное множество решений
  Нажмите любую клавишу для продолжения...
 ———————————— Test #4 ————————————
              Вычислить действительные корни квадратного уравнения:
                a*x*x + b*x + c = 0
  Введите значение a: (long) === > 0
  Введите значение b: (long) === > 2
  Введите значение c: (long) === > 222
                a = 0;  b = 2;  c = 222
  Уравнение линейное
                x=-111
  Нажмите любую клавишу для продолжения...
 ———————————— Test #5 ————————————
              Вычислить действительные корни квадратного уравнения:
                a*x*x + b*x + c = 0
  Введите значение a: (long) === > 768879
  Введите значение b: (long) === > -675785
  Введите значение c: (long) === > 786
                a = 768879;  b = -675785;  c = 786

  Результат в C++:
                d=454267994112
                b*b=456685355008;           4.*a*c=2417355520
                x1=0.877757787704467773     x2=0.00116464297752827406

  Результат в ASM:
                x1=0.877757787704467773     x2=0.00116463506128638983
                d=454268026880
                b*b=456685355008;           4.*a*c=2417355520

  Нажмите любую клавишу для продолжения...
 ———————————— Test #6 ————————————
              Вычислить действительные корни квадратного уравнения:
                a*x*x + b*x + c = 0
  Введите значение a: (long) === > -5676758
  Введите значение b: (long) === > 9678689
  Введите значение c: (long) === > 467
                a = -5676758;  b = 9678689;  c = 467

  Результат в C++:
                d=93687623188480
                b*b=93677019987968;               4.*a*c=-10604183552
                x1=-4.82409886899404228e-05       x2=1.70501589775085449

  Результат в ASM:
                x1=-4.82489704154431820-05        x2=1.70501589775085449
                d=93687623188480
                b*b=93677019987968;               4.*a*c=-10604183552

  Нажмите любую клавишу для продолжения...
```

 **ОБРАТИТЕ ВНИМАНИЕ**

В некоторых примерах точность результатов на С++ (особенно для небольших вещественных чисел) отличается от Ассемблера. Происходит это потому, что компилятор использует по умолчанию директиву процессора **.286р** (см. файл **pr13_3.asm** в материалах, прилагаемых к книге, или получите его сами, — не забыли, как это можно сделать?). Мы же в модуле на Ассемблере использовали директиву старшего сопроцессора **.387**. Кроме того, может быть и разная реализация трансцендентных функций в С++ и Ассемблере, но это тема отдельных исследований...

13.5.3. Организация разветвлений

Итак, мы знаем, что аналога знакового флага SF процессора в сопроцессоре нет. Но и беззнаковых данных в сопроцессоре тоже нет (см. табл. 13.1), поэтому наличие такого флага и не нужно. Специальных команд условного перехода у сопроцессора тоже нет, с ним прекрасно справляются уже изученные нами команды процессора.

Команды сравнения сопроцессора вырабатывают нужные нам флаги и, вспомнив, как мы организовывали условные переходы для беззнаковых данных в процессоре (см. табл. 9.3), соответственно преобразуем таблицы 13.11 и 13.12.

Таблица 13.16. Команды условного перехода при выполнении команд сравнения базового сопроцессора.

Команды условного перехода процессора	Условие	Флаги сопроцессора		
		C3	C2	C0
		Аналоги Flags (EFLAGS)		
		ZF	PF	CF
JA/JNBE	ST(0) > источник	0	0	0
JAE/JNB	ST(0) >= источник	1	0	0
JB/JNAE/JC	ST(0) < источник	0	0	1
JBE/JNA	ST(0) <= источник	1	0	1
JE/JZ	ST(0) = источник	1	0	0
JP	Операнды несравнимы	1	1	1

Теперь мы рассмотрим два способа организации разветвлений в сопроцессоре.

1. Способ, основанный на использовании команд сравнения базового сопроцессора (см. табл. 13.10), включает в себя следующие шаги:

```
; 1) Выделить память под слово состояния сопроцессора
        status    dw    ?
    ..........................................
; 2) Использовать нужную команду сравнения сопроцессора
;       .8087, .287, .387, .487, .587
        FCOMx, FICOMx, FTST
; 3) Переслать в память содержимое регистра SWR
```

```
        FSTSW     status
; 4) Организовать паузу, пока закончится пересылка
        FWAIT
; 5) Переслать в регистр AH содержимое
; СТАРШЕГО байта слова состояния сопроцессора
        MOV       AH,Byte PTR status+1
; 6) Получить флаги из регистра AH
        SAHF
; 7) Использовать нужную команду передачи управления
        Jx        Label
.........................................
Label:
```

2. Продвинутый способ, основанный на использовании команд сравнения процессора **Pentium Pro** (см. табл. 13.10) — более быстрый, но **не всем процессорам и компиляторам понятный**. Кроме того, работает ТОЛЬКО на 32-разрядных платформах.

```
; 1)Подключить процессор Pentium Pro или выше
.686p
; 2) Использовать нужную команду сравнения процессора
;         Pentium Pro или выше
        FCOMIx,   FUCOMIx
; 3) Использовать нужную команду передачи управления
        Jx        Label
.........................................
Label:
```

Недаром в народе говорят, что иная простота хуже воровства. Например, я смогла этот способ реализовать в **TASM32.exe версии 5.3** (он входит в стандартную поставку **Borland C++ Builder 5**) на процессоре **AMD Athlon-650**. Поэтому, прежде чем этот способ использовать, нужно убедиться, что ваши ресурсы его понимают. А это, на сегодняшний день, может свести на нет весь выигрыш от кажущейся простоты решения.

```
; 1) Проверка поддержки команд FCOMIx, FUCOMIx
;                        данным процессором
;(см. справочник В.Юрова [Л4-19], с. 184-187)
        mov       Eax,1
        CPUid
        bt        Edx,15          ; бит <15>=CF=1?
        JNC       no_FCOMI        ; НЕТ
; 2) Реализовать способ номер 2 организации разветвлений
;                       в сопроцессоре
```

```
.........................................
no_FCOMI:
; 3) Реализовать способ номер 1 организации разветвлений
;                        в сопроцессоре
.........................................
```

Впрочем, процессоры развиваются и становятся доступными массовому пользователю быстро. Так что вполне возможно, что через год-два надобность в такой предварительной проверке возможностей процессора отпадет...

ПРИМЕР 13.4.

Продолжим решение примера 13.3. Но теперь проверку дискриминанта **d=b²-4*a*c<0** будем проводить и в Ассемблере. Остальные проверки, чтобы не загромождать пример, делаются по-прежнему в C++. Любознательный и дотошный читатель, разобравшись с одной проверкой, вполне может сам сделать и остальные.

Решение. Применим для проверки дискриминанта способ № 1 и, соответственно 16-разрядную платформу MS DOS. В связи с этим немного видоизменится и сама программа на C++. **Добавленные команды, переменные и измененные фрагменты программы на C++ выделены жирным шрифтом.**

Решение этой задачи в связке Паскаль+Ассемблер вы можете найти в прилагаемых к данной книге материалах.

Исходный текст модуля Quadr4.asm

```
        title real8087  (CopyRight by Голубь Н.Г., 1993, 1998, 2001)
        ; 13.4. Пример программы на языке Ассемблера
        ; Вычислить действительные корни квадратного уравнения:
        ;          a*x*x + b*x + c = 0
        ;        TASM.EXE QUADR.ASM /l /r /ml
        ;  Особая ситуация d=b²-4*a*c<0 обрабатывается здесь
        ;——————————————————————————————————————————
        .387
        .model large,C
        .data
EXTRN  C  a:Dword,b:Dword,c:Dword,x1:dword,x2:dword,d:dword,f:byte
EXTRN C ac:Dword,bb:Dword
        .code
public          quadr
; Слово состояния сопроцессора
status          dw    ?
;——— КОНСТАНТЫ
four            dd    4.
two             dd    2.
quadr           proc C far
        finit                           ;иниц. 8087
                                        ;———ST(0)———   !  ———ST(1)———!
        fild    b                       ;b               !?
        fmul    st(0),st(0)             ;b*b             !?
        FST     bb                      ;копирование вершины стека ==> bb
        fild    a                       ;a               !b*b
        fmul    four                    ;4*a             !b*b
        fimul   c                       ;4*a*c           !b*b
```

```
        FST     ac                      ;копирование вершины стека ==> ac
        fsubP   st(1),st(0)             ;d=b*b-4*a*c          !?
        fst     d                       ;копирование вершины стека ==> d
        FTST                            ;проверка d=0
        FSTSW       status              ;слово состояния программы
        FWAIT
        mov     ah,byte ptr status+1
        sahf            ;ah->cf,pf,zf
        JB      im      ;d<0  (cf=1)
        fsqrt                           ;sqrt(d)              !?
        fld     st(0)                   ;sqrt(d)              !sqrt(d)
        fchs                            ;-sqrt(d)             !sqrt(d)
        fiadd   b                       ;b-sqrt(d)            !sqrt(d)
        fchs                            ;-b+sqrt(d)           !sqrt(d)
        fxch    st(1)                   ;sqrt(d)              !-b+sqrt(d)
        fiadd   b                       ;b+sqrt(d)            !-b+sqrt(d)
        fchs                            ;-b-sqrt(d)           !-b+sqrt(d)
        fidiv   a                       ;-b-sqrt(d)/a         !-b+sqrt(d)
        fdiv    two                     ;-b-sqrt(d)/a/2       !-b+sqrt(d)
        fstp    x2                      ;-b+sqrt(d)           !?
        fidiv   a                       ;-b+sqrt(d)/a         !-b+sqrt(d)
        fdiv    two                     ;-b+sqrt(d)/a/2 !?
        fstp    x1                      ;?                    !?
        mov     f,0     ; ok!           ;?                    !?
        ret
im:
        mov     f,1     ; d<0           ;d                    !?
        ret
quadr   endp
        end
```

Исходный текст программы Pr13_4.cpp

```
/* Pr13_4.cpp              Borland C++ 5.02
         Проект !!!!!! Application.exe === > DOS Standard !!!!!!
                              asm+cpp файлы
       (c) Copyright 1998-2001 by Голубь Н.Г.
                        Лекционный пример 13.4.
     Вычислить действительные корни квадратного уравнения:
                           a*x*x + b*x + c = 0
              Корректный ввод числовых символов.
                    Поддержка КИРИЛЛИЦЫ.
*/

#include <iomanip.h>
#include <conio.h>
#include <math.h>
#include "convert.h"  // Перекодировка КИРИЛЛИЦЫ Win—>DOS
#include "inputNum.h" // Ввод ЛЮБЫХ ЦЕЛЫХ чисел

const char* InputA = TXT(" Введите значение a: ");
const char* InputB = TXT(" Введите значение b: ");
const char* InputC = TXT(" Введите значение c: ");
const char* ResultC =   TXT("\n\n Результат в C++: ");
const char* ResultASM = TXT("\n\n Результат в ASM: ");
const char* CONTINUE = TXT("\n\n Нажмите любую клавишу для продолжения...");
const char* CALC =
```

```
        TXT("\n\tВычислить действительные корни квадратного уравнения:\n");
const char* UNCOUNTABLE = TXT("\nБесчисленное множество решений\n");
const char* LINEAR = TXT("\nУравнение линейное");
const char* NOROOTS = TXT("\n!!!!!!!! Нет вещественных корней !!!!!!!\n");
const char* NOASMRESULT=
       TXT("\n!!!!!!!! Корректное решение в Ассемблере НЕВОЗМОЖНО !!!!!!!\n");
//————————— ГЛОБАЛЬНЫЕ переменные — передаются в Ассемблер
long int a,b,c;
float x1,x2,d,ac,bb;
char f;
//————————————————————————————————————————————

// ВНЕШНИЕ ассемблерные функции
extern "C"
{
 void quadr (void);
}
 void title(void)
{
 cout << CALC;
 cout << "           a*x*x + b*x + c = 0\n";
}

 void title(long int a,long int b, long int c)
{
 cout <<"\n\t  a = "<< a << ";  b = " << b << ";  c = " << c ;
}

void PressAnyKey()
{ cout << CONTINUE;
   getch();
}

int quadrC(long int a, long int b, long int c);

void cond (long int a, long int b, long int c)
{
  cout << ResultC;
  f=0;  //ok!
  if (!a)//(a==0)
          if (!b)
    {
     cout << UNCOUNTABLE;
      f=1;
    }
          else
               { cout << LINEAR;
                  float x=-(float)c/b;
                 cout << "\n     x=" << x << endl;
                  f=2;
               }
  else
          if(quadrC(a,b,c))
            cout << NOROOTS;
          else
            cout << "\n x1=" << setprecision(20)<< x1 << "    x2=" << x2 << endl;
}
```

```
int quadrC(long int a, long int b, long int c)
{ bb=(float)b*b;
  ac=4.*a*c;
  d=bb-ac;
  cout  << "\n     d=" << setprecision(20)<< d << "\n   b*b=" << bb
        << ";  4.*a*c=" << ac << endl;
  if (d<0)return 1;
  else
       { x1=(-b+sqrt(d))/(2.*a);
         x2=(-b-sqrt(d))/(2.*a);
         return 0;
       }
}

       void main()
       { int t=0;
   for(;;) // бесконечный цикл
       {
          clrscr();
          cout << "——————————— Test #" << ++t << " ———————————";
          title();
          while (input (InputA,a));
          while (input (InputB,b));
          while (input (InputC,c));
          title (a,b,c);
          cond(a,b,c);
      //  if(!f)
            { x1=x2=0;
              d=bb=ac=f=0;
              cout << ResultASM;
              quadr();
              cout << "\n     d=" << setprecision(20)<< d << "\n   b*b=" << bb
                   << ";  4.*a*c=" << ac << endl;
              if (!f)
              cout << "\n x1=" << x1 << "   x2=" << x2 << endl;
              else  cout <<  NOROOTS;
            }
      //  else cout << NOASMRESULT;
          PressAnyKey();
      }
}
```

```
———————————— Test #1 ————————————
Вычислить действительные корни квадратного уравнения:
     a*x*x + b*x + c = 0
Введите значение a: (long) === > 11
Введите значение b: (long) === > -111
Введите значение c: (long) === > 1
     a = 11;  b = -111;  c = 1

Результат в C++:
     d=12277
     b*b=12321;   4.*a*c=44
x1=10.0818920135498047   x2=0.00901706609874963 76
```

```
Результат в ASM:
     d=12277
     b*b=12321;    4.*a*c=44
x1=10.0818920135498047    x2=0.00901706609874963 76
Нажмите любую клавишу для продолжения...
—————————— Test #2 ——————————
Вычислить действительные корни квадратного уравнения:
     a*x*x + b*x + c = 0
Введите значение a: (long) === > 1
Введите значение b: (long) === > -3
Введите значение c: (long) === > 2
     a = 1;  b = -3;  c = 2
Результат в C++:
     d=1
     b*b=9;    4.*a*c=8
x1=2  x2=1
Результат в ASM:
     d=1
     b*b=9;    4.*a*c=8
x1=2    x2=1
Нажмите любую клавишу для продолжения...
—————————— Test #3 ——————————
     Вычислить действительные корни квадратного уравнения
     a*x*x + b*x + c = 0
Введите значение a: (long) === > -67587857856
Ошибка! НЕ корректен диапазон для типа long
Введите значение a: (long) === > -785
Введите значение b: (long) === > 8967989780
Ошибка! НЕ корректен диапазон для типа long
Введите значение b: (long) === > 7568857
Введите значение c: (long) === > 1111111
     a = -785;  b = 7568857;  c = 1111111
Результат в C++:
     d=57291084660736
     b*b=57287594999808;    4.*a*c=-3488888576
x1=-0.146776497364044189    x2=9642.0029296875
Результат в ASM:
     d=57291084660736
     b*b=57287594999808;    4.*a*c=-3488888576
x1=-0.146798133850097656    x2=9642.0029296875
Нажмите любую клавишу для продолжения...
—————————— Test #4 ——————————
Вычислить действительные корни квадратного уравнения:
     a*x*x + b*x + c = 0
Введите значение a: (long) === > 1111
Введите значение b: (long) === > -1111
Введите значение c: (long) === > 111
```

```
     a = 1111;  b = -1111;  c = 111
Результат в C++:
     d=741037
     b*b=1234321;   4.*a*c=493284
x1=0.887414515018463135   x2=0.112585484981536865

Результат в ASM:
     d=741037
     b*b=1234321;   4.*a*c=493284
x1=0.887414515018463135   x2=0.112585484981536865

Нажмите любую клавишу для продолжения...
————————— Test #5 —————————
Вычислить действительные корни квадратного уравнения:
     a*x*x + b*x + c = 0
Введите значение a: (long) === > 0
Введите значение b: (long) === > 0
Введите значение c: (long) === > 0
     a = 0;  b = 0;  c = 0

Результат в C++:
Бесчисленное множество решений

Результат в ASM:
     d=0
     b*b=0;   4.*a*c=0
x1=-NAN   x2=-NAN

Нажмите любую клавишу для продолжения...
————————— Test #6 —————————
Вычислить действительные корни квадратного уравнения:
     a*x*x + b*x + c = 0
Введите значение a: (long) === > 0
Введите значение b: (long) === > 22
Введите значение c: (long) === > 222
     a = 0;  b = 22;  c = 222

Результат в C++:
Уравнение линейное
x=-10.0909090042114258

Результат в ASM:
     d=484
     b*b=484;   4.*a*c=0
x1=-NAN   x2=-INF

Нажмите любую клавишу для продолжения...
```

Вот мы и встретились с особыми числами... Надо от них избавиться. Для этого мы и ввели в программу на C++ **флажок f** и его анализ, связанный с проверкой исходных данных **a** и **b**.

Снимем комментарии в главной программе.

После внесения изменений в программу на C++ получим следующие тесты:

```
———————— Test #5 ————————
Вычислить действительные корни квадратного уравнения:
      a*x*x + b*x + c = 0
Введите значение a: (long) === > 0
Введите значение b: (long) === > 0
Введите значение c: (long) === > 0
      a = 0;  b = 0;  c = 0

Результат в C++:
Бесчисленное множество решений
!!!!!!!!  Корректное решение в Ассемблере НЕВОЗМОЖНО  !!!!!!!

Нажмите любую клавишу для продолжения...
———————— Test #6 ————————
Вычислить действительные корни квадратного уравнения:
      a*x*x + b*x + c = 0
Введите значение a: (long) === > 0
Введите значение b: (long) === > 111
Введите значение c: (long) === > 1111
      a = 0;  b = 111;  c = 1111

Результат в C++:
Уравнение линейное
      x=-10.009
!!!!!!!!  Корректное решение в Ассемблере НЕВОЗМОЖНО  !!!!!!!

 Нажмите любую клавишу для продолжения...
———————— Test #7 ————————
      Вычислить действительные корни квадратного уравнения:
      a*x*x + b*x + c = 0
 Введите значение a: (long) === > 8567786
 Введите значение b: (long) === > 0
 Введите значение c: (long) === > 7456867
      a = 8567786;  b = 0;  c = 7456867

 Результат в C++:
      d=-255555369172992
      b*b=0;   4.*a*c=255555369172992
!!!!!!!!  Нет вещественных корней  !!!!!!!

 Результат в ASM:
      d=-255555369172992
      b*b=0;   4.*a*c=255555369172992
!!!!!!!!  Нет вещественных корней  !!!!!!!

 Нажмите любую клавишу для продолжения...
```

Теперь вроде все считается нормально. Но не совсем...

А как вам понравятся такие тестовые примеры:

```
———————— Test #1 ————————
Вычислить действительные корни квадратного уравнения:
    a*x*x + b*x + c = 0
Введите значение a: (long) === > 234
Введите значение b: (long) === > -423342423
Введите значение c: (long) === > 234
    a = 234;  b = -423342423;  c = 234

Результат в C++:
    d=179218814779523072
    b*b=179218814779523072;   4.*a*c=219024
    x1=1809155.625   x2=-0.019351523369550705

Результат в ASM:
    d=179218814779523072
    b*b=179218814779523072;   4.*a*c=219024
    x1=1809155.625   x2=5.52744040760444477e-07

Нажмите любую клавишу для продолжения...
———————— Test #2 ————————
Вычислить действительные корни квадратного уравнения:
    a*x*x + b*x + c = 0
Введите значение a: (long) === > 22
Введите значение b: (long) === > 22222222
Введите значение c: (long) === > 22
    a = 22;  b = 22222222;  c = 22

Результат в C++:
    d=493827152412672
    b*b=493827152412672;   4.*a*c=1936
    x1=0.00091809616424143 3144   x2=-1010101

Результат в ASM:
    d=493827152412672
    b*b=493827152412672;   4.*a*c=1936
    x1=-9.8999998954241164e-07   x2=-1010101

Нажмите любую клавишу для продолжения...
———————— Test #3 ————————
Вычислить действительные корни квадратного уравнения:
    a*x*x + b*x + c = 0
Введите значение a: (long) === > 43
Введите значение b: (long) === > -4434344
Введите значение c: (long) === > 44
    a = 43;  b = -4434344;  c = 44

Результат в C++:
    d=19663406759936
    b*b=19663406759936;   4.*a*c=7568
    x1=103124.28125 x2=-6.50315123493783176e-05
```

```
Результат в ASM:
    d=19663406759936
    b*b=19663406759936;       4.*a*c=7568
    x1=103124.28125    x2=9.92254990705987439e-06
Нажмите любую клавишу для продолжения...
```

А это результат реализации Паскаль+Ассемблер:

```
———————— Test #1 ————————
Вычислить:   действительные корни квадратного уравнения:
    a*x*x + b*x + c = 0
введите a,b (longint):  234 -423342423
введите c (longint):  234
a=234;  b=-423342423;  c=234
    АССЕМБЛЕР:
    b*b= 1.79218814779523E+0017;    4*a*c= 2.19024000000000E+0005
    d=b*b-4*a*c= 1.79218814779523E+0017
    x1= 1.80915562500000E+0006; x2= 5.52744040760444E-0007
    ПАСКАЛЬ:
    b*b= 1.79218814779523E+0017;    4*a*c= 2.19024000000000E+0005
    d=b*b-4*a*c= 1.79218814779523E+0017
    x1= 1.80915562500000E+0006; x2=-1.93515233695507E-0002
продолжать? (y/n)
```

Такие вот парадоксы... В чем же тут дело? Кто же врет (C++, Паскаль, Ассемблер) и почему? Ответ оставляю за вами, мои уважаемые читатели. Думайте, исследуйте, пробуйте...

13.6. Машинные форматы команд сопроцессора

В главе 8 мы познакомились с машинными кодами базового процессора. Рассмотрим теперь машинные форматы команд сопроцессора, которые идейно похожи на команды процессора. Все команды сопроцессора в своем формате имеют вторичный код операции, поэтому минимальная длина команды сопроцессора 2 байта. Полный перечень кодов команд сопроцессора см. в прил. 9.

Сопроцессор имеет пять вариантов основных форматов команд. Во всех форматах (кроме сопроцессоров, встроенных в **Pentium Pro** и выше) минимальная длина собственно команды сопроцессора составляет 2 байта, а максимальная — 4 байта (добавляются 2 байта смещения для адресов). Кроме того, перед кодом операции может появиться префикс синхронизации команд сопроцессора **9Bh** (команда **FWAIT/WAIT**), а к смещению может быть добавлен сегментный префикс, если операнд находится в памяти НЕ в сегменте данных, точно так, как это происходит и в командах процессора.

Для сопроцессоров, встроенных в **Pentium Pro** и выше, префикс **9Bh** появляется только там, где это действительно необходимо. Смещения для адресов становятся 32-разрядными.

Код операции всех команд сопроцессора начинается с двоичного набора **11011** (ESC), который идентифицирует для процессора класс команд сопроцессора (вроде первой буквы **F** в командах сопроцессора для человека). Операнд в памяти можно указывать, используя любой режим адресации (см. табл. 8.3).

	Байт кода операции							
Биты	7	6	5	4	3	2	1	0
Кодировка	1	1	0	1	1	**MF**		1

	Байт способа адресации							
Биты	7	6	5	4	3	2	1	0
Кодировка	**mod**		КОП$_2$			**r/m**		

РИС. 13.6. *Вариант 1 — формат команд пересылки данных.*

Биты **MF** машинной команды задают тип операнда, находящегося в оперативной памяти (Memory Format):

00 — короткое вещественное число (32 бита);

01 — короткое целое число (32 бита);

10 — длинное вещественное число (64 бита);

11 — целое слово (16 бит) или упакованное десятичное число (80 бит).

Назначение битов **mod** и **r/m** в байте способа адресации — см. п. 8.1.2.

	Байт кода операции							
Биты	7	6	5	4	3	2	1	0
Кодировка	1	1	0	1	1	**MF**		0

	Байт способа адресации							
Биты	7	6	5	4	3	2	1	0
Кодировка	**mod**		КОП$_2$			**r/m**		

РИС. 13.7. *Вариант 2 — формат арифметических команд и команд сравнения с использованием операндов в оперативной памяти.*

	Байт кода операции							
Биты	7	6	5	4	3	2	1	0
Кодировка	1	1	0	1	1	**d**	**P**	0

	Байт способа адресации							
Биты	7	6	5	4	3	2	1	0
Кодировка	1	1	КОП$_2$		**R**	**ST(i)**		

РИС. 13.8. *Вариант 3 — формат арифметических команд и команд сравнения с использованием ST(i).*

Бит направления **d**. Если **d** = 0, то результат заносится в **ST(0)**, иначе — в **ST(i)**.

Бит **P** (Pop) указывает, производится ли после операции извлечение (выталкивание) результата из стека (**P=1**) или нет (**P=0**).

Бит **R** (Reverse) показывает реверсная ли данная команда (**R XOR d=1**) или нет (**R XOR d =0**). Этот бит имеет смысл рассматривать только для команд вычитания или деления.

	Байт кода операции							
Биты	7	6	5	4	3	2	1	0
Кодировка	1	1	0	1	1	0	0	1

	Байт способа адресации							
Биты	7	6	5	4	3	2	1	0
Кодировка	1	1	1	КОП$_2$				

РИС. 13.9. *Вариант 4 — формат трансцендентных и специальных арифметических команд, а также команд загрузки констант.*

	Байт кода операции										Байт способа адресации							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
Кодировка	1	1	0	1	1	0	1	1		Кодировка	1	1	1	КОП2				

РИС. 13.10. *Вариант 5 — формат команд управления сопроцессором.*

Проанализируем фрагмент **LST**-файла с машинными кодами из примера 13.3. В нем есть разнообразные команды сопроцессора, с которыми мы и разберемся.

```
Turbo Assembler  Version 4.1     17/09/01 23:13:15     Page 1
QUADR.ASM
real8087  (CopyRight by  Голубь Н.Г., 1993, 1998, 2001)
                      ; 13.3. Пример программы на языке Ассемблера
                      ; Вычислить действительные корни квадратного уравнения:
                      ;        a*x*x + b*x + c =0
                      ;        TASM.EXE QUADR.ASM   /l /r /ml
                      ; ВСЕ     особые ситуации    обрабатываются в С++
                      ;————————————————————————————————————————
                                          .387
*Warning* QUADR.ASM(8) Auxiliary processor incompatible with main processor
 0000                                     .model large,C
 0000                                   .data
                        EXTRN  C  a:Dword,b:Dword,c:Dword,x1:dword,x2:dword,d:dword
                        EXTRN C ac:Dword,bb:Dword
 0000                                  .code
                                       public  quadr
                              ;——— КОНСТАНТЫ
 0000  40800000                        four    dd      4.
 0004  40000000                        two     dd      2.
 0008                               quadr   proc C far
 0008  9B DB E3                        finit   ;инициализация 80387
                                               ;———ST(0)—————!——ST(1)———!
 000B  9B DB 06 0000e                  fild    b                ;b           !?
 0010  9B D8 C8                                fmul st(0),st(0) ;b*b         !?
 0013  9B D9 16 0000e                  FST     bb   ;копирование вершины стека ==> bb
 0018  9B DB 06 0000e                             fild    a           ;a           !b*b
 001D  9B 2E: D8 0E 0000r                         fmul   four         ;4*a         !b*b
 0023  9B DA 0E 0000e                             fimul   c           ;4*a*c       !b*b
 0028  9B D9 16 0000e                  FST     ac;копирование вершины стека ==> ac
                                    ;—————————————— альтернативные решения:
                                               ;fsubrp  st(1),st(0)   ;d=4*a*c-b*b  !?
                                               ;fsub    st(0),st(1)   ;d=4*a*c-b*b  !b*b
                                               ;fchs                  ;d=-4*a*c+b*b !?
                                    ;————————————————————————————————————
 002D  9B DE E9                                fsubP    st(1),st(0)  ;d=b*b-4*a*c !?
 0030  9B D9 16 0000e                          fst      d;копирование вершины стека ==> d
 0035  9B D9 FA                                fsqrt            ;sqrt(d)        !?
 0038  9B D9 C0                                fld      st(0)   ;sqrt(d)        !sqrt(d)
 003B  9B D9 E0                                fchs             ;-sqrt(d)       !sqrt(d)
 003E  9B DA 06 0000e                          fiadd  b         ;b-sqrt(d)      !sqrt(d)
 0043  9B D9 E0                                fchs             ;-b+sqrt(d)     !sqrt(d)
 0046  9B D9 C9                                fxch     st(1)   ;sqrt(d)        !-b+sqrt(d)
 0049  9B DA 06 0000e                          fiadd    b       ;b+sqrt(d)      !-b+sqrt(d)
 004E  9B D9 E0                                fchs             ;-b-sqrt(d)     !-b+sqrt(d)
```

```
   42    0051  9B DA 36 0000e        fidiv    a       ;-b-sqrt(d)/a      !-b+sqrt(d)
   43    0056  9B 2E: D8  36 0004r   fdiv     two     ;-b-sqrt(d)/a/2  !-b+sqrt(d)
   44    005C  9B D9 1E 0000e        fstp     x2      ;-b+sqrt(d)        !?
   45    0061  9B DA 36 0000e        fidiv    a       ;-b+sqrt(d)/a      !-b+sqrt(d)
   46    0066  9B 2E: D8  36 0004r   fdiv     two     ;-b+sqrt(d)/a/2   !?
   47    006C  9B D9 1E 0000e        fstp     x1      ;?                 !?
1  48    0071  CB              RET    00000h
   49    0072       quadr      endp
   50                                end
```

Проанализируем выделенные **жирным шрифтом** строки листинга. Обратите внимание, что **перед каждой командой сопроцессора** стоит префикс **9Bh.** Его мы при последующем разборе команд сопроцессора учитывать будем, но в описаниях соответствующих команд опустим (для экономии времени и места). Все коды операций команд сопроцессора приведены в прил. 9.

```
7                          387
```

***Warning* QUADR.ASM(8) Auxiliary processor incompatible with main processor**

Это предупреждение о том, что заявленный процессор (**i80386**) несовместим с главным процессором (в данном конкретном случае с процессором **AMD Athlon-650**).

```
18 0008   9B DB E3            finit      ;инициализация 80387
```

Данная команда (**DB E3**) относится к разряду управляющих — см. вариант 5. Вся информация о назначении этой команды хранится в битах 0-4 БСА ($КОП_2$).

	Байт кода операции										**Байт способа адресации**							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	0	1	1			1	1	1	0	0	0	1	1
Кодировка	1	1	0	1	1	0	1	1		Кодировка	1	1	1	$КОП_2$				

```
20 000B   9B DB 06 0000e      fild  b    ;b                 !?
```

Эта команда (**DB 06**) из группы команд пересылки данных — см. вариант 1. Распишем ее в двоичном виде и заполним соответствующие поля в байте кода операции и в байте способа адресации (БСА).

	Байт кода операции										**Байт способа адресации**							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	0	1	1			0	0	0	0	0	1	1	0
Кодировка	1	1	0	1	1	**MF**		1		Кодировка	**mod**		$КОП_2$			**r/m**		

MF=01 — пересылается 32-битное целое число. Анализируем БСА. Из табл. 8.3 получим: **mod=00**, значит, **r/m=110=смещение.** Имеем ***прямую адресацию.*** В машинной команде, кроме машинного кода операции, появляется еще и смещение с буквой **e** — **0000e** (**b** — внешняя переменная и описана, как полагается по умолчанию, в сегменте данных — см. п.8.2).

```
21 0010  9B D8 C8       fmul st(0),st(0) ;b*b          !?
```

Эта команда (**D8 C8**) из группы арифметических команд с использованием **ST(i)** — см. вариант 3. Распишем ее в двоичном виде и заполним соответствующие поля в байте кода операции и в байте способа адресации (БСА).

	Байт кода операции									Байт способа адресации							
Биты	7	6	5	4	3	2	1	0	Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	0	0	0		1	1	0	0	1	0	0	0
Кодировка	1	1	0	1	1	**d**	**P**	0	Кодировка	1	1	КОП$_2$		**R**	**ST(i)**		

Бит направления **d = 0** — результат заносится в **ST(0)**.

Бит **P=0** — после операции извлечение (выталкивание) результата из стека НЕ происходит.

Бит **R** для команды умножения смысла НЕ имеет.

```
22 0013 9B D9 16 0000e FST bb  ;копирование вершины стека ==> bb
```

Эта команда (**D9 16**) из группы команд пересылки данных — см. вариант 1. Распишем ее в двоичном виде и заполним соответствующие поля в байте кода операции и в байте способа адресации (БСА).

	Байт кода операции									Байт способа адресации							
Биты	7	6	5	4	3	2	1	0	Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	0	0	1		0	0	0	1	0	1	1	0
Кодировка	1	1	0	1	1	**MF**		1	Кодировка	**mod**		КОП$_2$			**r/m**		

MF=00 — операнд в памяти является 32-битным вещественным числом. Анализируем БСА. Из табл. 8.3 получим: **mod=00**, значит, **r/m=110=смещение**. Имеем *прямую адресацию*. В машинной команде, кроме машинного кода операции, появляется еще и смещение с буквой **e** — **0000e** (**bb** — переменная внешняя и описана в сегменте данных — см. п.8.2).

```
24 001D   9B 2E: D8 0E 0000r   fmul  four   ;4*a          !b*b
```

Эта команда (**D8 0E**) из группы арифметических команд с использованием операндов в оперативной памяти — см. вариант 2. Распишем ее в двоичном виде и заполним соответствующие поля.

	Байт кода операции									Байт способа адресации							
Биты	7	6	5	4	3	2	1	0	Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	0	0	0		0	0	0	0	1	1	1	0
Кодировка	1	1	0	1	1	**MF**		0	Кодировка	**mod**		КОП$_2$			**r/m**		

MF=00 — операнд в памяти является 32-битным вещественным числом. Анализируем БСА. Из табл. 8.3 получим: **mod=00**, значит, **r/m=110=смещение**. Имеем *прямую адресацию*. В машинной команде, кроме машинного кода операции, появляется еще и смещение с буквой **r** — **0000r**. Константа **four** описана в сегменте кода, поэтому в команде присутствует еще один префикс — префикс сегмента кода **2E:** (см. п.8.2).

```
25 0023   9B DA 0E 0000e       fimul    c       ;4*a*c              !b*b
```

Эта команда (**DA 0E**) практически аналогична предыдущей команде, за исключением формата операнда **c**. Распишем команду в двоичном виде и заполним соответствующие поля.

	Байт кода операции										Байт способа адресации							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	0	1	0			0	0	0	0	1	1	1	0
Кодировка	1	1	0	1	1	MF		0		Кодировка	mod		КОП$_2$			r/m		

MF=01 — операнд в памяти является 32-битным целым числом. Анализируем БСА. Из табл. 8.3 получим: **mod=00**, значит, **r/m=110=смещение**. Имеем *прямую адресацию*.

В машинной команде, кроме машинного кода операции, появляется еще и смещение с буквой **e** — **0000e** (**c** — переменная внешняя и описана, как полагается по умолчанию, в сегменте данных — см. п.8.2).

```
32 002D   9B DE E9          fsubP     st(1),st(0)    ;d=b*b-4*a*c  !?
```

Эта команда из группы арифметических команд с использованием **ST(i)** — см. вариант 3. Распишем ее (**DE E9**) в двоичном виде и заполним соответствующие поля в байте кода операции и в байте способа адресации (БСА).

	Байт кода операции										Байт способа адресации							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	1	1	0			1	1	1	0	1	0	0	1
Кодировка	1	1	0	1	1	d	P	0		Кодировка	1	1	КОП$_2$		R	ST(i)		

Бит **d = 1** — результат заносится в **ST(i)=ST(1)**.

Бит **P=1** — после операции происходит извлечение (выталкивание) результата из стека.

Бит **R** (Reverse) показывает реверсная ли данная команда (**R XOR d=1**) или нет (**R XOR d =0**). В нашем случае для команды вычитания получаем **1 XOR 1=0** — команда обычная, НЕ реверсная.

```
34 0035   9B D9 FA          fsqrt                    ;sqrt(d)     !?
```

Данная команда (**D9 FA**) совсем простая и относится к формату трансцендентных команд — см. вариант 4. Вся информация о назначении этой команды хранится в битах 0-4 БСА (КОП$_2$).

	Байт кода операции										Байт способа адресации							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	0	0	1			1	1	1	1	1	0	1	0
Кодировка	1	1	0	1	1	0	0	1		Кодировка	1	1	1	$КОП_2$				

```
35 0038    9B D9 C0        fld        st(0) ;sqrt(d)      !sqrt(d)
```

Эта команда, как и команда в строке 20, — из группы команд пересылки данных, но вариант 1 в данном случае здесь НЕ подходит, поскольку источником является регистр данных сопроцессора. Посмотрим на наиболее подходящий вариант 3. Распишем нашу команду (**D9 C0**) в двоичном виде и заполним соответствующие поля в байте кода операции и в байте способа адресации (БСА).

	Байт кода операции										Байт способа адресации							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	0	0	1			1	1	0	0	0	0	0	0
Кодировка	1	1	0	1	1	**d**	**P**	0		Кодировка	1	1	$КОП_2$		**R**	**ST(i)**		

Бит **d** = **0** — результат заносится в **ST(0)**. Мы знаем, что любая команда загрузки загружает **ST(0)**. Бит **P=0** — после операции НЕ происходит извлечения (выталкивания) результата из стека. Для команд загрузки выталкивания никогда НЕ производится — оно просто НЕ имеет смысла. Бит **R** для данной команды тоже смысла НЕ имеет. Таким образом, уважаемый читатель, мы встретились с НЕДОКУМЕНТИРОВАННЫМ форматом команды — поздравляю!

```
36 003B    9B D9 E0            fchs              ;-sqrt(d)    !sqrt(d)
```

Данная команда (**D9 E0**) очень простая и относится к формату специальных арифметических команд — см. вариант 4. Вся информация о назначении этой команды хранится в битах 0-4 БСА ($КОП_2$).

	Байт кода операции										Байт способа адресации							
Биты	7	6	5	4	3	2	1	0		Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	0	0	1			1	1	1	0	0	0	0	0
Кодировка	1	1	0	1	1	0	0	1		Кодировка	1	1	1	$КОП_2$				

```
39 0046    9B D9 C9            fxch st(1)        ;sqrt(d)     !-b+sqrt(d)
```

Эта команда, как и команда в строке 35, — из группы команд пересылки данных, где источником является регистр данных сопроцессора. Посмотрим на наиболее подходящий вариант 3. Распишем нашу команду (**D9 C9**) в двоичном виде и заполним соответствующие поля в байте кода операции и в байте способа адресации (БСА).

	Байт кода операции							
Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	0	0	1
Кодировка	1	1	0	1	1	**d**	**P**	0

	Байт способа адресации							
Биты	7	6	5	4	3	2	1	0
	1	1	0	0	1	0	0	1
Кодировка	1	1	КОП2		**R**	**ST(i)**		

Бит направления **d = 0** — результат заносится в **ST(0)**. Бит **P=0** — после операции НЕ происходит извлечения (выталкивания) результата из стека. Для нашей команды обмена выталкивания результата из стека никогда НЕ производится — оно просто НЕ имеет смысла. Бит **R** для данной команды тоже смысла НЕ имеет. Таким образом, уважаемый читатель, мы еще раз встретились с НЕДОКУМЕНТИРОВАННЫМ форматом команды.

```
44 005C   9B D9 1E 0000e       fstp      x2        ;-b+sqrt(d)      !?
```

Эта команда из группы команд пересылки данных — см. вариант 1 и почти полностью (с точностью до одного бита) совпадает с командой в строке 22. Распишем ее в двоичном виде и заполним соответствующие поля.

	Байт кода операции							
Биты	7	6	5	4	3	2	1	0
	1	1	0	1	1	0	0	1
Кодировка	1	1	0	1	1	**MF**		1

	Байт способа адресации							
Биты	7	6	5	4	3	2	1	0
	0	0	0	1	1	1	1	0
Кодировка	**mod**		КОП2			**r/m**		

MF=00 — операнд в памяти является 32-битным вещественным числом. Анализируем БСА. Из табл. 8.3 получим: **mod=00**, значит, **r/m=110=смещение**. Имеем ***прямую адресацию.*** В машинной команде, кроме машинного кода операции, появляется еще и смещение с буквой **e** — **0000e** (**x2** — переменная внешняя и описана в сегменте данных — см. п.8.2).

Из анализа двух команд **fld st(0)** и **fxch st(1)** совершенно четко видно, что должен быть известен еще формат для команд пересылки данных с использованием **ST(i)**. Вообще с сопроцессором связано довольно много интересных и недостаточно подробно изложенных в специальной литературе вопросов. Связано это с тем, что с сопроцессором на Ассемблере практически НЕ работают или работают очень мало.



Основы организации ввода-вывода информации

Мы можем предвидеть только то, что конструируем сами.

Людвиг Витгенштейн (1889–1951 гг.)

В этой главе мы расстанемся с нашими верными помощникам — компиляторами с алгоритмических языков Паскаль и С/С++, которые всю организацию ввода-вывода брали на себя, организовывая эти операции на так называемом **высоком уровне** (т.е. с помощью алгоритмических языков) и займемся сами на Ассемблере (т.е. на **низком уровне**) основными идеями и реализацией ввода-вывода информации.

Вернемся к архитектуре современного компьютера — см. п. 3.2. Мы знаем, что все устройства компьютера принято делить на внешние и внутренние. Внутренние устройства — это центральный процессор (ЦП) и оперативная память (ОЗУ), а внешние (ВЗУ) — все остальные. Обмен информации между ЦП и ВЗУ происходит через порты — это **самый нижний (аппаратный) уровень** ввода-вывода. Он — достаточно сложный и требует знания многих особенностей и технических деталей конкретной модели компьютера, которых мы в данной книге слегка коснемся. Следующий уровень ввода-вывода — на **уровне операционной системы (программный)**. Этот уровень практически НЕ зависит от технических деталей компьютера. Вот именно с этим, **программным уровнем** ввода-вывода мы и познакомимся сначала более подробно.

Особенности программного ввода-вывода информации всецело определяются **операционной системой**, под управлением которой выполняется прикладная программа, написанная на любом языке программирования. В функции любой современной операционной системы, поддерживающей выполнение прикладной программы, входит следующее:

- выделение программе области памяти для выполнения;
- загрузка программы;
- взаимодействие программы с устройствами ввода-вывода (портами), файловыми системами и другими программами.

Для выполнения системных программ (например, драйверов устройств) нужны и другие функции операционной системы. Мы их рассматривать НЕ будем, ограничившись классом прикладных программ, с которыми мы до сих пор работали.

Самой простой операционной системой для процессоров семейства Intel является дисковая операционная система (DOS) от разных производителей: Microsoft — MS DOS, IBM — PC-DOS, Novell — Novell DOS, Caldera — Open DOS. Она распространяется как сама по себе, так и в виде части операционных систем линейки Microsoft Windows. DOS предоставляет программисту **полную свободу доступа** к оперативной памяти, устройствам, портам, никак не ограничивая его свободу действий. Поэтому при изучении основ языка Ассемблера она так популярна. С нее мы и начнем.

Базой программного ввода-вывода (далее просто ввода-вывода) является понятие **прерывание** (*interrupt*) и **подпрограмма обработки прерываний** (**ISR** — interrupt service routine), которая анализирует событие, достойное внимания операционной системы. Вкратце происходит это следующим образом. Процессор, получив от операционной системы специальный сигнал (сигнал прерывания), приостанавливает выполнение команд текущей программы и передает управление операционной системе. Она определяет, какое событие произошло (нажата клавиша на клавиатуре или мышке, символ попал в область видеопамяти дисплея и проч.) и реагирует на него, вызывая соответствующую подпрограмму обработки прерываний. Вызов нужного прерывания происходит по команде **INT** (INTerrupt - прерывание). Существуют различные требования для указания операционной системе, какое действие (ввод или вывод) и на каком устройстве необходимо выполнить. Программа (и программист) при этом могут детально НЕ знать, как это сделать. Нужно просто сформировать нужную информацию и сгенерировать подходящий тип прерывания. Мы в данной главе сосредоточимся на основных прерываниях и требованиях для вывода информации на экран и ввода данных с клавиатуры. Поняв, как это делается, нетрудно будет изучить и более изощренные операции ввода-вывода.

14.1. Исполняемые программы в MS DOS

Известно, что **исполняемые прикладные программы** в DOS имеют два основных формата: COM- и EXE-файлы. Мы даже немного поэкспериментировали с EXE-файлом в п. 5.1.1.3. Операционная система DOS при загрузке исполняемых файлов ориентируется на первые два байта их заголовка. Для EXE-файлов - это символы MZ или ZM и еще некоторые внутренние признаки, которые мы и рассмотрим более подробно.

Для получения исполняемого файла нужно сначала вызвать компилятор с языка Ассемблера, как это мы уже неоднократно делали, а затем сделать вызов **соответствующей версии компоновщика (tlink** — для **TASM, link16** и, если нужно, **exe2bin** — для **MASM).**

Утилита **exe2bin** позволяет при соблюдении **всех** необходимых условий (см. п 14.1.1) получить из EXE-формата COM-файл:

Exe2bin Myfile.EXE Myfile.COM

При всех прочих равных условиях COM-файл всегда будет меньше, чем соответствующий EXE-файл.

14.1.1. Характеристика COM-файла

1. Длина исполняемого файла ограничена размером одного сегмента и не может превышать **64 K** байт.
2. Нет сегмента стека и сегмента данных. В программе все данные должны быть определены в сегменте кода.
3. Все процедуры, включая главную, должны иметь атрибут **NEAR**.
   Поэтому в современных программах на Ассемблере для COM-файлов используют следующую модель (см. п. 3.4.2):
   .model TINY
4. Нужно с помощью директивы **ORG 100h** установить значение начального смещения **100h**, поскольку DOS резервирует память длиной в 256 байт (100h) для **PSP** (Prefix Segment Program — префикса программного сегмента).
5. Не нужно специально готовить стек для нормального возврата в DOS — это делает сама операционная система.
6. Все данные и резервируемую для работы программы память нужно размещать в виде блока либо в начале программного кода, либо в конце, поскольку программа НЕ имеет сегмента данных. Чтобы повысить эффективность вычислений, рекомендуется данные и резервирование памяти размещать ДО выполняемого кода и через команду **JMP** делать переход непосредственно на выполняемую часть программы.
7. Для получения исполняемого COM-файла ПОСЛЕ вызова компилятора с языка Ассемблера нужно вызвать **компоновщик.**

Например, для получения файлов **MyFile.lst, MyFile.obj, MyFile.com** при использовании компилятора **TASM** нужно выполнить следующие вызовы:

tasm /l MyFile.asm
tlink MyFile.obj /t


ЗАМЕЧАНИЕ 1.

Обратите внимание на ключ **/t** при вызове компоновщика **tlink.**

То же - при использовании компилятора **MASM**:
ml /AT /Fl MyFile.asm
link16 MyFile.obj,,NUL,,,
Exe2bin Myfile.EXE Myfile.COM


ЗАМЕЧАНИЕ 2.

Попытка выполнить файл **Myfile.EXE** приведет к возникновению ошибки, поскольку этот файл **является промежуточным** для получения исполняемого файла **Myfile.COM.**

Несоблюдение хотя бы одного требования COM-формата может служить причиной неправильной работы программы, поэтому будьте внимательны.

Для облегчения компиляции и компоновки COM-файлов в среде TASM сделаем командный файл:

Командный файл buildCOM.bat

```
@echo off
if -%1 == - goto NoParam
if -%2 == - goto NoParam
if not exist %1 goto NotExist
goto Ok
:NotExist
echo ERROR!!!
type %1
echo Not Exist!!!
goto exit
:NoParam
echo ERROR!!!
echo Variant of a call:
echo builtCOM.bat Rewri.asm Rewri.com
goto exit
:Ok
tasm %1 temp.obj /la
if exist %2 del %2
tlink temp.obj /t
ren Temp.com %2
if exist *.obj del *.obj >nul
if exist *.map del *.map >nul
:exit
```

Вызов этого файла предполагает указание двух имен файлов: **существующего ASM-файла** и нужного COM-файла, например,

builtCOM.bat Rewri.asm Rewri.com

В случае несоответствия формату вызова или ошибки вызова этого командного файла будет выдано соответствующее диагностическое сообщение. Кроме того, возможны ошибки компиляции и компоновки. С ошибками компиляции проще — для этого мы делаем LST-файл. Чтобы увидеть все остальные ошибки, нужно использовать командную оболочку типа широко известного FAR, поскольку выполнение этого файла будет происходить в окне DOS.

Схематически общую структуру COM-файла можно изобразить следующим образом (рис. 14.1).

```
.model   TINY
.CODE
ORG 100h
; Точка входа в программу
start:        jmp   main
;─────────────────────────────
; Данные и резервируемая память
;─────────────────────────────
;                      Процедуры
;───────────────────────────────
;──────── Главная программа
main proc
;   Тело главной программы
main endp
;─────────────────────────────
;                      Процедуры
;───────────────────────────────
   end  start  ; Указание точки входа обязательно
```

РИС. 14.1. *Общая структура COM-файла*

Для COM-файла DOS автоматически определяет стек и устанавливает одинаковый общий сегментный адрес во всех четырех сегментных регистрах (CS, DS, ES, SS). Если для программы размер сегмента в 64K является достаточным, то DOS устанавливает в регистре SP адрес конца сегмента - 0FFFEh. Это будет вершина стека. Если 64K байтовый сегмент не имеет достаточно места для стека, то DOS устанавливает вершину стека в конце оперативной памяти, выделенной программе. Затем в обоих случаях DOS записывает в стек нулевое слово.

Таким образом, возможность использования стека зависит от размера программы и размера выделенной для программы памяти.

14.1.2. Характеристика EXE-файла

1. Длина исполняемого файла НЕ ограничена размером одного сегмента.
2. Есть сегмент стека, и может быть сегмент данных, если это необходимо. Обычно данные располагаются в сегменте данных.
3. Главная процедура и все остальные процедуры могут иметь атрибут NEAR или FAR в зависимости от используемой модели памяти (см. п. 3.4.2).
4. EXE-файл имеет стандартный заголовок размером в 512 байт — об этом заботится сам компилятор.
5. Нужно специально готовить стек для нормального возврата в DOS — см. п. 14.3.1.1.
6. Нужно делать инициализацию сегмента данных DS программы — см. п. 14.3.1.1.
7. Для получения исполняемого EXE-файла ПОСЛЕ вызова компилятора с языка Ассемблера тоже нужно вызвать **компоновщик.**

Например, для получения файлов **MyFile.lst, MyFile.obj, MyFile.EXE** при использовании компилятора **TASM** нужно выполнить следующие вызовы:

tasm /l MyFile.asm
tlink MyFile.obj

 **ЗАМЕЧАНИЕ.**

Обратите внимание **на отсутствие** ключа **/t** при вызове компоновщика **tlink.**

То же - при использовании компилятора **MASM**:
ml /Fl MyFile.asm
link16 MyFile.obj,,NUL,,,

Сделаем и здесь командный файл для облегчения компиляции и компоновки EXE-файлов в среде TASM. Он будет немного отличаться от приведенного выше файла buildCOM.bat:

Командный файл buildEXE.bat

```
@echo off
if -%1 == - goto NoParam
if -%2 == - goto NoParam
if not exist %1 goto NotExist
goto Ok
```

```
:NotExist
echo ERROR!!!
type %1
echo Not Exist!!!
goto exit

:NoParam
echo ERROR!!!
echo Variant of a call:
echo builtEXE.bat Rewri.asm Rewri.exe
goto exit

:Ok
tasm %1 temp.obj /la
if exist %2 del %2
tlink temp.obj
ren Temp.exe %2
if exist *.obj del *.obj >nul
if exist *.map del *.map >nul
:exit
```

EXE-файл может иметь самую разнообразную структуру в зависимости от используемой модели памяти. Изобразим схематически один из вариантов общей структуры EXE-файла для модели Large — см. рис. 14.2.

```
.model  LARGE
.stack
; Данные в стеке
.data
;---------------------------
; Данные и резервируемая память
;---------------------------
.CODE
; Точка входа в программу
main proc FAR
;----------------------------
; Тело главной программы
;----------------------------
main  endp
;---------------------------
;                        Процедуры
;----------------------------
   end  main   ; Указание ТОЧКИ ВХОДА ОБЯЗАТЕЛЬНО!!!
```

РИС. 14.2. *Общая структура EXE-файла для модели Large (один из возможных вариантов).*

14.2. Команды обработки прерывания INTx

Любая операция ввода-вывода в DOS может быть реализована через определенную подпрограмму обработки прерывания (ISR). Для вызова этих подпрограмм служат следующие команды семейства INTx:

INTO

INT 3

INT *НомерПрерывания*

Логика работы команды INTO:

1) Если флаг переполнения **OF=0**, то никаких дополнительных действий НЕ производить и продолжить выполнение программы.

2) Если флаг переполнения **OF=1**, то инициировать команду **INT 4**.

Логика работы команды INT:

1) Записать в стек регистр флагов **EFLAGS (FLAGS)**.

2) Записать в стек адрес возврата:
   - содержимое сегментного регистра **CS**;
   - содержимое указателя команд **EIP (IP)**.

3) Сбросить в ноль флаг **IF** (и **TF**, если делается отладка или вызвано прерывание **INT** 3).

4) Передать управление программе обработки прерываний с указанным номером *НомерПрерывания*. Для этого используется специальная системная область памяти — **таблица векторов прерываний** (см. прил. 6).

5) Обеспечить выполнение необходимых действий.

6) Восстановить из стека адрес возврата и флаги и продолжить выполнение прерванной программы.

Все необходимые экранные и клавиатурные операции можно выполнить, используя соответствующие функции команды **INT 10h** (вывод на дисплей - видеосервис) и **INT 16h** (ввод с клавиатуры), которые передают управление непосредственно в **BIOS** (*Basic Input Output System* — базовая система ввода-вывода). BIOS любого компьютера семейства i86 находится в ПЗУ (одна или две микросхемы), установленном на системной (материнской) плате. Современные компьютеры имеют еще и дополнительные ПЗУ BIOS, расположенные на картах расширений (например, на видеокартах), которые содержат альтернативные ISR для обработки вывода на экран.

Для выполнения некоторых более сложных операций существует прерывание более высокого уровня **INT 21h**, которое сначала передает управление в DOS. Например, при вводе с клавиатуры может потребоваться подсчет введенных символов, проверка на максимальное число символов и проверка на символ Enter. Прерывание DOS **INT 21h** выполняет многие из этих дополнительных вычислений и затем автоматически передает управление в BIOS.

Таким образом, команда **INT** прерывает обработку программы, передает управление в DOS и/или BIOS для выполнения определенных действий и затем возвращает управление в прерванную программу для продолжения дальнейших вычислений.

ЗАМЕЧАНИЯ

- При реализации операций ввода-вывода через прерывания BIOS нужно иметь в виду, что многие функции могут быть модифицированы BIOS данного компьютера. Поэтому тот результат, на который вы рассчитывали при отладке программы на вашем компьютере, может не получиться на другом.
- Подпрограммы обработки прерываний в процессе своей работы могут портить регистры. Поэтому рекомендуется до вызова прерывания сохранить в стеке нужные регистры, а затем восстановить их и продолжить выполнение программы.

14.3. Основные функции сервисного прерывания MS DOS 21h

Чтобы обратиться к нужной функции DOS, нужно выполнить следующие действия:

- Загрузить регистры, согласно описанию функции.
- Подготовить необходимые буферы, строки ASCII и проч.
- Поместить номер функции DOS в регистр **AH**. Если используется функция с подфункцией, то номер подфункции обычно помещается в регистр **AL**.
- Вызвать прерывание DOS **INT 21h.**

Например, для завершения программы с определенным *кодом Code* и передачи управления DOS нужно выполнить функцию **4Ch** (см. пример 14.5):

```
MOV      AL, Code
MOV      AH,4Ch
INT      21h
```

14.3.1. Вывод информации на дисплей

В табл. 14.1 перечислены наиболее употребительные функции прерывания DOS 21h, которые выводят **информацию** на дисплей в **символьном виде** в коде ASCII (см. табл. П2.1 и П2.3 — для символов кириллицы).

Таблица 14.1. Основные функции вывода информации на дисплей для прерывания DOS 21h.

Регистр AH	**Операция**	**Дополнительные входные регистры или данные**
2	Изображение символа с проверкой на **Ctrl-Break**.	<DL>=Символ
6	Изображение символа без проверки на **Ctrl-Break**.	<DL>=Символ
9	Изображение строки символов с проверкой на **Ctrl-Break**.	<DS:DX>- начальный адрес строки. **Символ окончания строки '$'.**

Проверка на **Ctrl-Break** означает, что если в процессе работы данной функции была нажата эта комбинация клавиш, то по умолчанию происходит прерывание 23h и выход из программы.

14.3.1.1. Вывод строк символов



ПРИМЕР 14.1

Сделать исполняемую программу в двух форматах (COM- и EXE-файлы), которая выводит сообщение: **'Ура! Вывод строки на дисплей в Ассемблере!!!'**.

Решение.

1. Строка или символ в Ассемблере записываются в ОДИНАРНЫХ кавычках ' '.
2. Воспользуемся функцией **9** прерывания **21h**.
3. Строка вывода будет иметь имя **message**. Заканчиваться она должна символом **'$'**.

Возможные варианты решения этой задачи следующие.

Исходный текст программы helloCOM.asm

```
            .Model      TINY  ; модель памяти, используемая для COM-файла
            .Code
; резервирование памяти для префикса программного сегмента (PSP)
            ORG         100h
Start: jmp    SHORT main    ; обход данных
; ————————— ДАННЫЕ —————————
; !!!!! сегментов ДАННЫХ, СТЕКА, ДОПОЛНИТЕЛЬНОГО НЕТ !!!!!!!!
;                         строка вывода
message       db     'Ура! Вывод строки на дисплей в Ассемблере!!!$'
;========= НАЧАЛО ПРОГРАММНОГО КОДА
main    proc    NEAR  ; ТОЛЬКО NEAR !!!! Иначе зависание при выполнении
   lea    dx,message        ; адрес строки - в DX
   mov    ah,9              ; номер функции DOS - в AH
   int    21h               ; вызов прерывания DOS
   ret                      ; завершение COM-программы
main      endP
   end    start  ; указание ТОЧКИ ВХОДА в программу start ОБЯЗАТЕЛЬНО!!!
```

Посмотрим теперь, чем будет **отличаться** реализация в формате EXE-файла.

Исходный текст программы helloEXE.asm

```
   .model  LARGE       ; одна из моделей памяти, используемая для EXE-файла
   .stack 100h         ; сегмент стека размером в 256 байт
   .data
message         db     'Ура! Вывод строки на дисплей в Ассемблере!!!$'
   .code
start  proc   FAR      ; FAR- для ГЛАВНОЙ ПРОЦЕДУРЫ (Модель large!!!)
; FAR обеспечивает нормальный вызов из DOS и DEBUG
;———————— 1. Подготовка НОРМАЛЬНОГО возврата в DOS ————————
      push      ds     ; DS использует ЗАГРУЗЧИК DOS  (адрес начала PSP)
      xor       di,di  ; нулевой адрес
      push      di     ;                        возврата в DOS
;————————2. Инициализация сегмента DS ——————————————
      mov       ax,@DATA
      mov       ds,ax
      lea       dx,message
      mov       ah,9       ; функция DOS "вывод строки"
      int       21h
      ret
start   endp
     end        start
```

Если теперь воспользоваться нашими BAT-файлами, то мы получим COM- и EXE-файлы. Чтобы что-то увидеть, надо запускать их, например, из оболочки FAR. Программы эти еще НЕ совершенны, т.к. мы не умеем пока организовывать паузу и ждать нажатия клавиши на клавиатуре, чтобы увидеть результат своих действий. Но скоро мы и этому научимся — см. пример 14.3.

14.3.1.2. Вывод целых чисел

Вывод целых чисел немного более сложный и состоит из двух этапов:

- преобразование внутреннего представления числа в строку символов (код ASCII);
- вывод полученной строки символов.

Если внимательно посмотреть на символьное представление цифр **'0',...,'9'** в коде ASCII (см. табл. П2.1), то можно заметить, что они в шестнадцатеричном виде имеют коды **30h,...,39h**. На этом и построен алгоритм преобразования. Мы остановимся на организации вывода целых чисел. Вывод **вещественных чисел** на порядок сложнее, как и само его внутреннее представление (см. п. 2.3), хотя **идеи преобразования одни и те же**.

Алгоритм преобразования целых (16-разрядных) чисел со знаком достаточно прост и сводится к последовательному делению числа (в его внутреннем, машинном, представлении) на константу 10 (поскольку нас интересует, конечно, десятичное представление числа). В процессе целочисленного деления получаются остатки, которые преобразуются в символьный вид и заносятся в буфер вывода.

1. Исходное целое число [-32768,...,32767] поместить в регистр AX.
2. Очистить буфер вывода (например, занести символ пробела) для последующего размещения в нем строкового представления числа: ЦЦЦЦЦ или —ЦЦЦЦЦ (где Ц — цифра {0,...,9}). Таким образом, **минимально возможная длина буфера составляет 6 символов**.
3. Все последующие арифметические операции (пп. 5-8) производятся с положительным числом. Поэтому нужно проверить знак числа.
4. Если число отрицательное, то сделать его положительным.
5. Выполнить целочисленное БЕЗЗНАКОВОЕ деление числа на 10.
6. ОСТАТОК от целочисленного деления перевести в символьное представление. Для этого достаточно остаток сложить с символом **'0'** (**30h** — см. табл. П2.1).
7. Занести символьное представление остатка в буфер вывода (остатки будут заноситься в буфер, начиная с конца буфера).
8. Проверить частное. Если оно НЕ равно НУЛЮ, перейти на п. 5.
9. Если исходное число было отрицательным, то занести в буфер вывода символ '-'.
10. Конец работы алгоритма.

Поясним этот алгоритм (пп. 3-9) на примере и сделаем **фрагмент подпрограммы Bin_ASCII** на Ассемблере.

ПРИМЕР 14.2

Пусть нужно преобразовать в строку число <AX>=—157.

3) Проверяем знак числа -157:

OR AX,AX

4) Оно отрицательное, делаем его положительным:

NEG AX

5) Делим наше ПОЛОЖИТЕЛЬНОЕ число (157) на 10 (константа 10 пусть хранится в регистре SI):

Clr_Dvd: ; начало цикла последовательного деления ПОЛОЖИТЕЛЬНОГО числа

XOR DX, DX

DIV SI

Получаем: частное <AX>=15, остаток <DX>=7

6) Переводим остаток от целочисленного деления (7) в символьное представление:

ADD DX, '0'

Получаем <DX>=37h

7) заносим символьное представление остатка ('7') в буфер вывода (см. рис. 14.3):

DEC BX

MOV Byte PTR [BX], DL

8) Проверяем частное на равенство его НУЛЮ:

OR AX, AX

JNZ Clr_Dvd

Наше частное <AX>=15 не равно нулю, поэтому переходим опять на п.5 и опять повторяем все пункты. В результате получаем следующее: частное <AX>=1, остаток <DX>=5. Преобразуем остаток в символ, получим <DX>=35h и опять заносим его ('5') в буфер вывода (см. рис. 14.3).

Теперь наше новое частное <AX>=1 тоже не равно нулю, поэтому переходим опять на п.5 и повторяем все пункты. В результате получаем следующее: частное <AX>=0, остаток <DX>=1. Преобразуем остаток в символ, получим <DX>=31h и опять заносим его ('1') в буфер вывода (см. рис. 14.3).

Теперь мы получили частное <AX>=0, поэтому процесс последовательного деления на 10 прекращаем.

9) Поскольку наше исходное число было отрицательным, то в буфер вывода нужно занести еще знак минус:

DEC BX

MOV Byte PTR [BX], '-'

10) На этом алгоритм заканчивается.

Символьное представление	' '	' '	'-'	'1'	'5'	'7'	'$'
Hex-код	20	20	2d	31	35	37	24

РИС. 14.3. *Внутреннее представление буфера вывода для числа —157.*

А теперь сделаем программу в COM-формате с подпрограммами **Bin_ASCII** и **Output** (чтобы можно было использовать их и в дальнейшем) и проверим ее работу, воспользовавшись результатами примера 14.1 и добавив в главную программу вывод еще и положительного числа.

Исходный текст программы WRIteCOM.ASM

```
Title Number (com-файл)
;CopyRight by Голубь Н. Г., 1993, 2001
; =========== COM - формат ==========================
;  TASM <file> [.asm]                    ====== > <file>.OBJ
;  TLINK <file> [.obj]/t                ====== > <file>.COM
;=====================================================
        .Model   TINY
        .Code
        ORG        100h ; резервирование памяти для PSP
Start:  jmp        SHORT main
; ─────── -L==-+ ────────
CR_LF   db         0dh,0ah,'$'
L       dw         6
Number  dw         12345
inv     db         'Демонстрация вывода чисел: $'
OutBuf  db         11 dup(?),'$'   ; Буфер вывода
;───────────────────
main    proc       NEAR
        lea        dx,inv
        call       output          ; вывод строки inv
        Lea        bx,OutBuf       ; адрес буфера вывода
        Mov        ax,Number

        CALL       Bin_ASCII       ; Binary -> ASCII
        Lea        dx,[bx]
        call       output          ; вывод ПОЛОЖИТЕЛЬНОГО числа

        Mov        ax, -157
        CALL       Bin_ASCII       ; Binary -> ASCII
        Lea        dx,[bx]
        call       output          ; вывод ОТРИЦАТЕЛЬНОГО числа
        lea        dx, CR_LF
        call       output          ; переход на след. строку
        ret
main    endp
; ───── вывод строки ────────────
; ВХОД
;    DX- начальный адрес строки.
;                 символ окончания строки '$'.
; ВЫХОД
;    НЕТ
;───────────────────
output  proc
        push ax
        mov  ah,9
        int  21h
        pop  ax
        ret
output  endp
```

```
;——— Procedure Bin_Ascii ——————————
; Вход:
;       число ——> AX
;       нач. адрес буфера вывода ——> BX
;       длина буфера вывода ——> L
; Выход:
;          строка цифр в виде -цццццц или цццццц
;          нач. адрес буфера вывода ——> BX
;          факт. длина буфера вывода ——> CX <= L
;——————————————————————————————
Bin_Ascii     Proc        Near
              Push        dx si ax
              Mov         cx,L

 Fill_Buf:            ;========= очистка буфера вывода
              Mov         Byte ptr [bx],' '
              Inc         bx
              Loop        Fill_Buf
;================= и установка BX на конец буфера
              Mov         si,10
              Or          ax,ax          ; c?
              Jns         Clr_Dvd          ; НЕТ, обход
              Neg         ax               ; ДА - меняем знак
 Clr_Dvd:
              Xor         dx,dx
              Div         si               ; <DX:AX> = <DX:AX>/10
              Add         dx,'0'           ; остаток —> ASCII
              Dec         bx
       ; символ —> буфер
              Mov         Byte PTR [bx],dl
              Inc         cx               ; кол-ва символов
              Or          ax,ax            ; частное = 0 ?
       ; НЕТ - продолжаем вычисления
              Jnz       Clr_Dvd

              Pop         ax ; выгружаем из стека исходное число
              Or          ax,ax            ; число ОТРИЦАТЕЛЬНОЕ ?
              Jns         NoMore           ; НЕТ, обход
              Dec         bx
       ; добавляем знак '-'
              Mov         Byte ptr [bx],'-'
              Inc         cx
 NoMore:
              Pop       si dx
              Ret
Bin_Ascii   Endp
          end       start
```

Запустив нашу программу на счет (к сожалению, чтобы увидеть результат, надо пока опять воспользоваться какой-либо текстовой оболочкой типа FAR), мы получим следующий результат:

```
Демонстрация вывода чисел: 12345      -157
```



ЗАМЕЧАНИЕ

Поскольку наша процедура **output** портит регистр AX, то перед вызовом процедуры обработки прерывания мы содержимое этого регистра сохраняем в стеке, а затем, по окончании обработки прерывания восстанавливаем его. Аналогичные действия мы сделали и в процедуре **Bin_Ascii.** Этот прием мы будем использовать и в дальнейших наших примерах, чтобы процедуры можно было использовать повторно.

14.3.2. Ввод информации с клавиатуры

Любая информация с клавиатуры с использованием соответствующих функций прерывания *DOS 21h* вводится тоже в **символьном виде** (в коде ASCII - см. *табл. П2.1* и *П2.3* — для символов кириллицы).

Таблица 14.2.
Основные функции ввода информации с клавиатуры для прерывания DOS 21h.

Регистр AH (Hex)	Операция	Дополнительные входные регистры или данные	Выходные регистры или данные
1	Ожидание набора символа с последующим его эхо-сопровождением и с проверкой на **Ctrl-Break**.		<AL>=Символ
7	Ожидание набора символа без последующего его эхо-сопровождения и без проверки на **Ctrl-Break**.		<AL>=Символ
8	Ожидание набора символа без последующего его эхо-сопровождения и с проверкой на **Ctrl-Break**.		<AL>=Символ
A	Чтение строки в буфер клавиатуры.	<DS:DX>- адрес буфера. Первый байт буфера – длина буфера.	Второй байт буфера – фактическая длина введенных в буфер символов.
B	Чтение состояния клавиатуры.		<AL>=0FFh, если буфер клавиатуры пуст. <AL>=0, если буфер клавиатуры НЕ пуст.
C	Освобождение буфера клавиатуры и вызов нужной функции.	<AL>= Номер нужной функции	В соответствии с вызываемой функцией

14.3.2.1. Организация ввода символов и строк в буфер клавиатуры

Один символ можно ввести так, чтобы он отображался на дисплее при вводе (эхо-сопровождение), или был невидимым (без эхо-сопровождения). Для этого используются функции 1, 7 или 8. Их программирование обычно никаких трудностей не вызывает. Запишем процедуру **readKey** ожидания ввода символа без эхо-сопровождения, чтобы в наших примерах делать задержку перед закрытием окна MS DOS (чего мы не умели делать при решении примеров 14.1 и 14.2).

```
;------- ввод символа БЕЗ эхо-сопровождения ---------
; ВХОД:
;       НЕТ
; ВЫХОД:
;       AL   - введенный символ
;-------------------------------------------------------
readKey proc
        push ax                ; сохранение регистра AX
        mov  ah,8              ; чтение символа
        int  21h
        pop  ax                ; восстановление регистра AX
        ret
readKey endp
```

Добавим теперь эту процедуру и **ее вызов** в файл WRIteCOM.ASM и получим вывод наших чисел с ожиданием нажатия любой клавиши:

```
Демонстрация вывода чисел: -12345      31157
Нажмите любую клавишу.......
```

Фрагмент дополненного исходного текста программы WRIteCOM.ASM

```
OutBuf   db   11 dup(?),'$'    ; Буфер вывода
pause    db   'Нажмите любую клавишу.......$'
; ---------------------------------------------

main     proc      NEAR
         lea  dx,inv
         call output              ; вывод строки inv

         Lea  bx,OutBuf           ; адрес буфера вывода
         Mov  ax,Number
         CALL Bin_ASCII           ; Binary ---> ASCII

         Lea      dx,[bx]
         call output              ; вывод ОТРИЦАТЕЛЬНОГО числа
         Mov  ax, 31157
         CALL Bin_ASCII           ; Binary ---> ASCII
         Lea  dx,[bx]
         call output              ; вывод ПОЛОЖИТЕЛЬНОГО числа
         lea  dx, CR_LF
         call output              ; переход на след. строку

         lea  dx,pause
         call output              ; вывод строки pause
         call readKey             ; ожидание ввода символа
         ret
main     endp
```

В нашем примере мы НЕ анализируем, какую именно клавишу на клавиатуре нажал пользователь.



ЗАМЕЧАНИЕ

Попробуйте поэкспериментировать, на какие клавиши наша функция **readKey** НЕ будет реагировать.

Если используется номер функции **0Ah**, то символ или строка вводятся в буфер клавиатуры, структура которого имеет вид, представленный на рис. 14.4.

Первый байт	Второй байт	Остальные L байт	
Общая длина буфера	Число фактически введенных в буфер символов	actL введенных символов	Свободно
L	actL	actL <= L	

РИС. 14.4. *Структура буфера клавиатуры.*

Пусть мы имеем буфер клавиатуры **Buffer** длиной **L=30** байт. На языке Ассемблера это можно записать следующим образом:

```
; Второй и остальные 30 байт заполнены пробелом
Buffer DB 30,31 dup (' ')
```

Или можно применить более структурированную (и понятную) запись, применив директиву **Label**:

```
;--- структура буфера ------
Buffer  LABEL     BYTE
L       db        30              ; максимальная длина
actL    db        ?               ; реальная длина
field   db        30   dup(?)     ; инф. поле буфера
```

В последнем случае можно обращаться как ко всему буферу (по имени **Buffer)** целиком, так и к именованным полям буфера: **L, actL** или **field**.



ПРИМЕР 14.3

Написать на языке Ассемблера программу ввода строки символов (не более 10 символов) и вывода ее на дисплей.

Будем решать эту задачу по шагам и тоже для двух форматов исполняемых файлов.

Шаг 1. Воспользуемся процедурой **output** вывода строки символов из реализации примера 14.2.

Шаг 2. Написать процедуру ввода символов в буфер клавиатуры - **readString**.

Воспользуемся функцией **0Ch** и подфункцией **0Ah** (ввод символов в буфер клавиатуры с предварительной очисткой буфера).

```
;—— ввод строки символов ——————
; ВХОД:
;     BUFFER - буфер клавиатуры
; ВЫХОД:
;     CX - фактическое число введенных символов
;     DX - адрес информационной части буфера
;     ACTL   - реальная длина буфера
;————————————————
readString proc
     push    ax ; сохранение регистра AX
        lea  dx,buffer
        mov      ah,0ch              ; очистка буфера
        mov      al,0ah              ; чтение строки в буфер
        int  21h
        xor      ch,ch
; реальное количество введенных символов
        mov      cl,actL
; установка указателя на информационную часть буфера
        add      dx,2
   pop  ax ; восстановление регистра AX
   ret
readString endp
```

Шаг 3. Все необходимые процедуры ввода-вывода у нас есть, теперь соберем все вместе и напишем программы целиком. При выводе строк на дисплей воспользуемся символами **0Dh** (Возврат каретки **CR**) и **0Ah** (Перевод строки **LF**) — см. табл. П2.2. **Buffer** будем использовать как для ввода строки символов, так и для вывода.

Исходный текст программы Rewricom.asm

```
title Read Write String com-файл
;CopyRight by Голубь Н.Г., 1993, 2001
; ============ COM - формат ==================================
;  TASM <file> [.asm]           ====== > <file>.OBJ
;  TLINK <file> [.obj]/t        ====== > <file>.COM
; ============================================================
      .Model  TINY
      .Code
      ORG     100h      ; резервирование памяти для
                        ; префикса программного сегмента (PSP)
Start: jmp    SHORT main    ; обход данных и экономия 1 байт
; ————————— ДАННЫЕ ——————————————
; !!!!! сегментов ДАННЫХ, СТЕКА, ДОПОЛНИТЕЛЬНОГО НЕТ !!!!!!
CR_LF   db     0dh,0ah,'$'
     ;— структура буфера ———
buffer LABEL  BYTE
maxL     db     11                    ; максимальная длина
actL     db     ?                     ; реальная длина
field    db      11  dup(?),'$'       ; поле буфера
     ; реальная длина 10 символов, т.к. ENTER (09h) тоже символ
inv      db     'Вводите, что хотите (10 символов) ———> $'
pause    db     'Нажмите любую клавишу.......$'
;——————————————————————————————
main     proc   NEAR          ; ТОЛЬКО NEAR !!!! Иначе зависание
         lea    dx,inv
   call  output                       ; вывод приглашения
```

```
    call    readString
    push    dx  ; запоминание в стеке адреса информационной части буфера
    lea     dx,CR_LF
    call    output                  ; переход на след. строку
    pop     dx  ; восстановление из стека адреса информационной части буфера
    call    output                  ; вывод введенных символов
    lea     dx, CR_LF
    call    output                  ; переход на след. строку
    lea     dx,pause
    call    output                  ; вывод строки pause
    call    readKey                 ; ожидание ввода символа
    ret
main    endp
; ———— вывод строки ————————————————————
; ВХОД:
;   DX- начальный адрес строки.
;           Символ окончания строки '$'.
; ВЫХОД:
;   НЕТ
;——————————————————————————————————————
output      proc
    push ax
        mov     ah,9
        int     21h
    pop  ax
        ret
output      endp
; ———— ввод строки символов ———————————————
; ВХОД:
;   BUFFER - буфер клавиатуры
; ВЫХОД:
;   CX - фактическое число введенных символов
;   DX - адрес информационной части буфера
;   ACTL   - реальная длина буфера
;——————————————————————————————————————
readString proc
    push    ax
        lea     dx,buffer
            mov     ah,0ch          ; очистка буфера
            mov     al,0ah          ; чтение строки в буфер
        int     21h
        xor     ch,ch
        mov     cl,actL             ; реальное количество введенных символов


        add     dx,2                ; уст. указателя на инф. часть буфера
    pop  ax
      ret
readString endp
;—————————— ввод символа БЕЗ эхо-сопровождения ————
; ВХОД:
;   НЕТ
; ВЫХОД:
;   AL   - введенный символ
;——————————————————————————————————————
readKey proc
```

```
    push ax                     ; сохранение регистра AX
         mov  ah,8              ; чтение символа
         int  21h
    pop  ax                     ; восстановление регистра AX
         ret
readKey endp
    end  start                  ; указание точки входа ОБЯЗАТЕЛЬНО!!!
```

EXE-файл реализуем немного по-иному, чем в примере 14.1, воспользовавшись для описания сегментов директивами SEGMENT. **В этом случае инициализация сегмента данных будет тоже другая.**

Фрагмент исходного текста программы Rewri.asm

```
title Read Write String EXE-файл
;CopyRight by Голубь Н.Г., 1993, 2001
;============== EXE - формат =====================================
;  TASM <file> [.asm]         ====== > <file>.OBJ
;  TLINK <file> [.obj]        ====== > <file>.EXE
; ==============================================================
;———————— сегмент СТЕКА ——————————————————————————
sstack  segment para stack 'stack'
      db     64 dup('stack')
sstack  ends
;———————— сегмент ДАННЫХ ——————————————————————————
dseg    segment byte
CR_LF   db     0dh,0ah,'$'
    ;— структура буфера ———
buffer LABEL  BYTE
maxL     db    11             ; максимальная длина
actL     db    ?              ; реальная длина
field    db    11  dup(?),'$' ; поле буфера
    ; реальная длина 10 символов, т.к. ENTER (09h) тоже символ
inv      db    'Вводите, что хотите (10 символов) ———> $'
pause    db    'Нажмите любую клавишу.......$'
dseg     ends

;———————— сегмент КОДА ——————————————————————————
code    segment para 'code'
      assume  cs:code,ds:dseg,ss:sstack
start  proc   FAR              ; для ГЛАВНОЙ ПРОЦЕДУРЫ
; FAR обеспечивает нормальный вызов из DOS и DEBUG
;————————————— подготовка НОРМАЛЬНОГО возврата в DOS ——————————
   push  ds            ; DS использует ЗАГРУЗЧИК (адрес начала PSP)
   xor   di,di         ; нулевой адрес
   push  di            ;          возврата в DOS
   mov   ax,dseg
   mov   ds,ax         ; инициализация DS программы
   lea   dx,inv
   call  output        ; вывод приглашения на ввод
   call  readString    ; ввод строки
   push  dx            ; запоминание в стеке адреса начала инф. части буфера
   lea   dx, CR_LF
   call  output        ; переход на след. строку
   pop   dx
   call  output        ; вывод введенных символов
   lea   dx, CR_LF
```

```
        call    output              ; переход на след. строку
        lea     dx,pause
        call    output              ; вывод строки pause
        call    readKey             ; ожидание ввода символа
        ret
;       RET     при NEAR вызывает ЗАЦИКЛИВАНИЕ      !!!!!!!!!!!!!!!!!
start   endp
;———————————————— вывод строки ————————————————————
.....................................................................
readKey endp
code    ends
        end     start           ; указание точки входа ОБЯЗАТЕЛЬНО!!!
```

Шаг 4. Получив соответственно COM- и EXE-файлы и запустив их на счет, мы увидим, что буфер ввода нужно заполнять до конца, чтобы получить на дисплее верный результат. В противном случае результирующая строка может быть или в усеченном виде, или вообще отсутствовать:

```
Вводите, что хотите (10 символов) ------> 1234

Нажмите любую клавишу.......
Вводите, что хотите (10 символов) ------> 1234567
   4567
Нажмите любую клавишу.......
Вводите, что хотите (10 символов) ------> 1234567890
1234567890
Нажмите любую клавишу.......
Вводите, что хотите (10 символов) ------> qwert 123
 wert 123
Нажмите любую клавишу.......
```

Шаг 5. Таким образом, идея использования буфера **buffer**, как для ввода строки, так и для ее вывода, НЕ подходит для общего случая. Сделаем **отдельный буфер вывода OutBuf** такого же размера, как и буфер ввода, воспользовавшись реализацией примера 14.2:

```
OutBuf  db       11 dup(?),'$'
```

и используем строковые операции пересылки (см. п. 11.1). Изменим только файл **Rewricom.asm**. В результате **главная процедура main изменится**, а остальные процедуры останутся прежними.

Фрагмент измененного файла Rewricom.asm

```
            .Model  TINY
            .Code
      ORG 100h ; резервирование памяти для префикса программного сегмента (PSP)
Start: jmp     SHORT main      ; обход данных и экономия 1 байт
; ——————————— ДАННЫЕ ———————————————————————
; !!!!! сегментов ДАННЫХ, СТЕКА, ДОПОЛНИТЕЛЬНОГО НЕТ !!!!!!
CR_LF   db      0dh,0ah,'$'
      ;— структура буфера ввода ———
buffer  LABEL   BYTE
maxL    db      11                      ; максимальная длина
actL    db      ?                       ; реальная длина
field   db      11  dup(?)              ; поле буфера
      ; реальная длина 10 символов, т.к. ENTER (09h) тоже символ
OutBuf  db      11 dup(?),'$'           ; Буфер вывода
inv     db      'Вводите, что хотите, - 10 символов ———> $'
pause   db      'Нажмите любую клавишу.......$'
;————————————————————————————————————————————
```

```
main    proc    NEAR            ; ТОЛЬКО NEAR !!!! Иначе зависание
        push    cs
        pop     es
        lea     dx,inv
        call    output          ; вывод приглашения
        call    readString
        push    dx cx
        lea     dx,CR_LF
        call    output          ; переход на след. строку
        pop     cx dx
        MOV     SI,dx           ; адрес информационной части буфера ввода
        LEA     DI,OutBuf       ; адрес буфера вывода
        REP     MOVSB           ; пересылка введенных символов
        Lea     dx,OutBuf
        call    output          ; вывод введенных символов
        lea     dx, CR_LF
        call    output          ; переход на след. строку
        lea     dx,pause
        call    output          ; вывод строки pause
        call    readKey         ; ожидание ввода символа
        ret
main    endp
```

Шаг 6. Протестируем полученный COM-файл:

```
Вводите, что хотите (10 символов) ------> 12345
12345
Нажмите любую клавишу.......
Вводите, что хотите (10 символов) ------> 1
1
Нажмите любую клавишу.......
Вводите, что хотите (10 символов) ------> 123456789
123456789
Нажмите любую клавишу.......
Вводите, что хотите (10 символов) ------> qwert 1234
qwert 1234
Нажмите любую клавишу.......
Вводите, что хотите (10 символов) ------> qwe 12
qwe 12
Нажмите любую клавишу.......
```

Теперь все нормально. Любознательный и дотошный читатель может САМ аналогичные действия проделать с файлом Rewri.asm.



ЗАМЕЧАНИЕ.

Если сразу не получится, посмотрите **внимательно** п. 11.1.

14.3.2.2. Ввод целых чисел

Ввод чисел немного более сложный. Он, как и вывод чисел, тоже состоит из двух этапов:

- ввод строки символов в буфер клавиатуры, в результате чего получится **символьное** представление числа в кодах ASCII — см. рис. 14.5;
- преобразование строки символов в коде ASCII во внутреннее представление числа.

Алгоритм преобразования строки символов во внутреннее представление ЦЕЛОГО **числа** будет обратным алгоритму, описанному в п. 14.3.1.2. Но, в отличие от него, он будет и более сложным. Поэтому поясним сначала **основную идею** алгоритма на примере.

ПРИМЕР 14.4.

Описать алгоритм получения внутреннего представления числа 359.

После ввода числа 359 в информационной части буфера клавиатуры сформируется строка, показанная на рис. 14.5.

Символы	'3'		'5'		'9'	
HEX-код	3	3	3	5	3	9
Полубайты	Старший	Мл.	Старший	Мл.	Старший	Мл.

РИС. 14.5. *Содержимое информационной части буфера клавиатуры при вводе числа 359.*

Заметим, что **содержимое младших полубайт — это и есть нужные нам составляющие числа**. Вспомним, что десятичная система счисления является позиционной (см. п. 1.1). Значит, наше число может быть представлено следующим образом:

$359 = 3*10^2 + 5*10^1 + 9*10^0$

На этих идеях и базируется **основной алгоритм** преобразования строки символов во внутреннее представление целого ПОЛОЖИТЕЛЬНОГО числа.

1. **Выделить старшую цифру числа (младшие четыре бита — младший полубайт) и занести в промежуточный результат.**

   Промежуточный результат = 3.

2. **Выделить следующую цифру числа. Если у числа цифр больше нет, перейти к п. 6.**

   Следующая цифра числа 5.

3. **Умножить промежуточный результат на число 10.**

   3*10=30.

4. **Добавить выделенную в п.2 цифру к произведению и результат занести в промежуточный результат.**

   Промежуточный результат = 5 + 30 = 35.

5. **Перейти к п. 2.**

Начинается *вторая итерация* нашего алгоритма:

2-2) Следующая цифра числа 9.

3-2) 35*10 = 350.

4-2) Промежуточный результат = 9 + 350 = 359.

5-2) Перейти к п. 2.

Начинается *третья итерация* нашего алгоритма:

2-3) Следующая цифра числа ОТСУТСТВУЕТ. Переходим к п.6.

6. **Конец работы алгоритма.** Промежуточный результат — это и есть внутреннее представление нашего числа.

В этом алгоритме не учтены следующие основные моменты:

- **Число может быть как положительным, так и отрицательным.** Это решается достаточно просто путем анализа символа знака. И если введено число отрицательное (присутствует ПЕРЕД числом символ '-'), то надо это обстоятельство запомнить, а число рассматривать как положительное. Затем вспомнить о знаке, как это мы уже делали в п. 14.3.1.2.
- **ПЕРЕД числом могут быть введены незначащие пробелы.** Значит, нужно предусмотреть операцию игнорирования этих пробелов (пропуск ведущих пробелов в буфере ввода).
- **По ошибке могут быть введены НЕ числовые символы.** Значит, нужно анализировать выделяемые в пп. 1 и 2 нашего **основного алгоритма** цифры. Если в результате анализа выяснится, что была введена НЕ цифра, то предусмотреть выдачу соответствующего сообщения.
- **Число может превысить допустимый диапазон.** В этом случае нужно предусмотреть проверку полученного нами числа на допустимый диапазон [-32768,..., 32767]

Теперь можно конструировать наши процедуры: **Ascii_Bin** — главная процедура посимвольного преобразования строки во внутреннее представление числа, которая вызывает процедуру **Conv_St** (реализация нашего алгоритма преобразования). Продемонстрируем эти процедуры и их использование на конкретном примере.

 **ПРИМЕР 14.5.**

Написать программу корректного ввода-вывода любого целого 16-разрядного числа. Реализуем программу в виде EXE-файла. Опишем наши действия по шагам.

Шаг 1. Процедура **Conv_St**. Сделаем нашу процедуру несколько избыточной, чтобы ее можно было повторно использовать в других программах.

```
;---------- Procedure Conv_St ----------------------
; посимвольное преобразование строки во внутреннее
; представление числа
; ВХОД:
;       нач. адрес строки в буфере ввода -----> BX
;       ERR2 - сообщение "НЕ цифра!"
; ВЫХОД:
;       число ------> AX
;           если введена не цифра,
;            то CF=1, DI <---- адрес ошиб. символа,
```

```
;                 флаг E=2
;-----------------------------------------------------
Conv_St      Proc       Near
                Push     bp
                Push     bx
                Push     si
                Mov      bp,bx
                Xor      bx,bx
                Mov      si,10
 Range:;===== проверка на допустимый символ [0,...,9]
                Cmp      Byte  ptr ds:[bp],'0'
                Jb        NoneDigit               ; < '0'
                Cmp      Byte  ptr ds:[bp],'9'
                Jbe      Digit                    ; <= '9'
 NoneDigit:   ;======= НЕ цифра
                Lea      dx,Err2
                Call     Output
                Mov      E,2
                Mov      di,bp; адрес ошиб. символа ---> DI
                Stc                        ; CF=1
                Jc       End_Conv
 Digit:       ;--------- обработка цифры
                Push     dx
                Mul      si            ; <DX:AX> = AX*10
                Pop      dx
                Mov      bl,ds:[bp]    ; ASCII - код
                And      bx,0fh         ; выделение МЛ. части
                Add      ax,bx
                Jc       End_Conv      ; если результат велик
                Inc      bp
                Loop     Range
                Clc                         ; CF=0
 End_Conv:
                Pop      si
                Pop      bx
                Pop      bp
                Ret
Conv_St      Endp
```

Шаг 2. Процедура **Ascii_Bin**. Поскольку все преобразования мы делаем с положительными числами, то и сравнения тоже будем делать БЕЗЗНАКОВЫМИ, чтобы немного **расширить диапазон** проверяемых чисел.

```
;---------- Procedure Ascii_Bin --------------------
; ВХОД:

;       нач. адрес строки --------> BX
;       размер буфера ввода ------> L
;       фактическое число введенных символов   ------> CX
;        сообщения об ошибке:
;          ERR1 - "Введено слишком много символов!"
;          ERR3 - "Нарушен диапазон!"
; ВЫХОД:
;        число ------> AX и X
;        если ошибок НЕТ, то CF=0, DI=0FFh
;               иначе  CF=1, DI <---- адрес ошиб. символа
```

```
;                    если введено > L символов, то E=1
;                    если нарушен диапазон, то E=3
;----------------------------------------------------
Ascii_Bin  Proc      Near
               Push     bx
               Push     cx
               Xor      ax,ax
               Mov      di,0ffh        ; ошибок НЕТ
; проверка количества введенных символов
               Cmp      cx,L
               Ja       Bad              ; CX > L
 BlanKs:   ;======== пропуск ведущих пробелов
               Cmp      Byte  Ptr [bx],' '
               Jne      ChkNeg          ; пробелов НЕТ
               Inc      bx
               Loop     Blanks
 ChkNeg:
       ;----------- число ОТРИЦАТЕЛЬНОЕ ?
               Cmp      Byte  ptr [bx],'-'
               Jne      ChkPos                  ; НЕТ
               Inc      bx                       ; след. символ
               Dec      cx
    ; преобразование ОТРИЦАТЕЛЬНОГО числа
               Call     Conv_St
    ; есть ошибки при преобразовании строки
               Jc       Quit
    ; проверка допустимого диапазона
               Cmp      ax,32768
               Ja       BadRange    ; AX > 32768 (WORD!!!)
               Neg      ax            ; МЕНЯЕМ ЗНАК
               Clc                    ; CF=0 (НЕТ ошибок)
               Js       Good
 ChkPos: ;------ преобразование ПОЛОЖИТЕЛЬНОГО числа
       ; есть знак '+'?
               Cmp      Byte  ptr [bx],'+'
               Jne      Go_Conv                 ; НЕТ
               Inc      bx        ; пропуск знака '+'
               Dec      cx
 Go_Conv:
               Call     Conv_St
   ; есть ошибки при преобразовании строки
               Jc       Quit
   ; проверка допустимого диапазона
               Cmp      ax,32767    ; AX > 32767 (WORD!!!)
               Ja       BadRange
 Good:    ;---------------------------- OK!
               Clc
               Jnc      Quit
;=========== Ошибки ========================
 Bad:   ;=== CX > L (Введено слишком много символов)
               Lea      dx,Err1
               Call     Output
               Mov      E,1
               Stc                          ; CF=1
               Jmp      Quit
```

```
BadRange: ;===== выход за диапазон
            Lea      dx,Err3
            Call     Output
            Mov      E,3
            Stc                          ; CF=1
 Quit:
            Pop      cx
            Pop      bx
            Mov      X,ax
            Ret
Ascii_Bin  Endp
```

Шаг 3. Сделаем немного отличный от примера 14.3 вариант EXE-файла (БЕЗ главной процедуры), чтобы показать разнообразие структур EXE-формата. Процедура **Main** будет осуществлять вызов процедур, нужных нам для решения задачи. **ReadKeys** — аналог процедуры **readString**. В конце задачи будем спрашивать у пользователя, желает ли он продолжения решения задачи. Реализуем здесь ввод символа с использованием функции 1 прерывания 21h и анализ введенного символа. Остальные процедуры нам уже известны. **Все новые строки кода выделим жирным шрифтом.**

Фрагменты исходного текста программы R_Wnumbr.asm

```
Title   R_Wnumbr.asm
;CopyRight by Голубь Н.Г., 1993, 2001
SStack      Segment    Byte    Public
              db     64 dup   ('stack')
SStack      Ends
;-------------------------------------------------
Data         Segment    Byte    Public
   Buffer     db 30,31 dup (' ')   ; Буфер ввода
   X            dw (?)
   L            dw 6   ; размер буфера
   E            db (?)      ; флаг ошибки
CR_LF    db 0dh,0ah,'$'
Meet1    db ' Введите X и нажмите <Enter>',0dh,0ah,'$'
Meet     db ' (-32768 < X < 32767) = $'
Result   db ' Assembler : X = $'
Repeat   db ' Продолжим ? ("N" - выход):$'
Err1     db ' Ошибка при вводе - введено слишком много символов! $'
Err2     db ' Введена НЕ цифра! $'
Err3     db ' Нарушен диапазон $'
GoodBye  db ' Good bye my friend',0dh,0ah,'$'
Res      db 6 dup (?),'$'; Буфер вывода
Data         EndS
;-------------------------------------------------
Code         Segment    Byte    Public
               Assume   cs:code,ds:data
Start:
               Call     Main
               Mov      ax,4c00h
               Int      21h

;---- Procedure Main ---------------------------------
```

```
Main          Proc      Near
                 Mov       ax,data
                 Mov       ds,ax
  Again:
                 Mov       E,0
                 Lea       dx,Meet1
                 Call      Output
                 Lea       dx,Meet
                 Call      Output
                 Call      ReadKeys
                 Push      dx
                 Lea       dx,CR_LF
                 Call      Output
                 Lea       dx,CR_LF
                 Call      Output
                 Pop       dx
                 Mov       bx,dx
                 Call      Ascii_Bin
                 Cmp       E,0
                 Jg        Erro
                 Lea       dx,Result
                 Call      Output
                 Lea       bx,Res
                 Call      Bin_Ascii
                 Lea       dx,[bx]
                 Call      Output
  Erro:
                 Lea       dx,CR_LF
                 Call      Output
                 Lea       dx,CR_LF
                 Call      Output
; продолжать ли решение задачи?
                 Lea       dx,Repeat
                 Call      Output
                 Mov       ah,1
                 Int       21h
                 Cmp       al,'n'
                 Je        Bye
                 Cmp       al,'N'
                 Je        Bye
                 Jmp       Again
  Bye:
                 Lea       dx,CR_LF
                 Call      Output
                 Lea       dx,GoodBye
                 Call      Output
                 Ret
Main          Endp
;---------- Procedure Output ------------------------
Output        Proc      Near
                 Push      ax
                 Mov       ah,9
                 Int       21h
                 Pop       ax
                 Ret
```

```
Output      Endp
;---------- Procedure ReadKeys ----------------------
ReadKeys    Proc      Near
                Push     ax
                Lea      dx,Buffer
                Mov      ah,0ch    ; очистка буфера
                Mov      al,0ah    ; чтение строки в буфер
                Int      21h
                Xor      ch,ch
; фактическое число введенных символов
                Mov      cl, Buffer+1
; указатель --> начало информации
                Add      dx,2
                Pop      ax
                Ret
ReadKeys    Endp
;---------- Procedure Ascii_Bin ---------------------
.......................................................
;---------- Procedure Conv_St -----------------------
.......................................................
;---------- Procedure Bin_Ascii ---------------------
; ВХОД:
;      число -------> AX
;      нач. адрес буфера ------> BX
; ВЫХОД:
;         строка цифр в виде -ЦЦЦЦЦ или ЦЦЦЦЦЦ
;         нач. адрес ------> BX
;         факт. длина буфера ------> CX <= 6
;---------------------------------------------------
Bin_Ascii   Proc      Near
                Push     dx
                Push     si
                Push     ax
                Mov      cx,6
 Fill_Buf:   ;=========== очистка буфера
                Mov      Byte ptr [bx],' '
                Inc      bx
                Loop     Fill_Buf
;=========== и установка BX на конец буфера

                Mov      si,10
                Or       ax,ax          ; число ОТРИЦАТЕЛЬНОЕ
                Jns      Clr_Dvd            ; НЕТ
                Neg      ax                  ; ДА - берем МОДУЛЬ
 Clr_Dvd:
                Xor      dx,dx
                Div      si             ; <DX:AX> = <DX:AX>/10
                Add      dx,'0'       ; остаток -----> ASCII
                Dec      bx
       ; символ -----> буфер
                Mov      Byte PTR [bx],dl
                Inc      cx                    ; кол-во символов
                Or       ax,ax                ; частное = 0 ?
                Jnz      Clr_Dvd              ; НЕТ
                Pop      ax          ; выгружаем исх. число
                Or       ax,ax     ; число ОТРИЦАТЕЛЬНОЕ ?
```

```
                Jns      NoMore              ; НЕТ

                Dec      bx
                Mov      Byte ptr [bx],'-'
                Inc      cx
  NoMore:
                Pop      si
                Pop      dx
                Ret
Bin_Ascii   Endp
CODE         EndS
                End      Start
```

Шаг 4. Проверим реализацию нашей задачи.

```
Введите X и нажмите <Enter>
[-32768 <= X <= 32767]=====> 11111
Assembler : X = 11111

Введите X и нажмите <Enter>
[-32768 <= X <= 32767]=====> -22222
Assembler : X = -22222

Введите X и нажмите <Enter>
[-32768 <= X <= 32767]=====> 3.333
Введена НЕ цифра!
Продолжим ? ("N" - выход):

Введите X и нажмите <Enter>
[-32768 <= X <= 32767]=====> 73gg3
Введена НЕ цифра!
Продолжим ? ("N" - выход):

Введите X и нажмите <Enter>
[-32768 <= X <= 32767]=====> 100000
Нарушен диапазон
Продолжим ? ("N" - выход):

Введите X и нажмите <Enter>
[-32768 <= X <= 32767]=====> 44444
Нарушен диапазон
Продолжим ? ("N" - выход):

Введите X и нажмите <Enter>
[-32768 <= X <= 32767]=====> 99999
Нарушен диапазон
Продолжим ? ("N" - выход):n
Good bye my friend
```



ЗАМЕЧАНИЕ

Проверьте **сами** решение нашей задачи (*Подвергай все сомнению!*). Здесь есть кое-что, на что надо обратить внимание. Предоставляю это вам, уважаемые мои читатели. Ибо чувствовать себя умным, знающим и понимающим — очень приятно...

14.4. Основные функции работы с экраном. Прерывание BIOS 10h

Рассмотренные нами функции вывода на экран с использованием прерывания DOS 21h не дают возможности вывести текст в любую позицию экрана, очистить экран, не дают изменить цвет текста. Все это позволяют выполнить видеофункции BIOS. Кроме того, BIOS позволяет работать дисплею, как в текстовом режиме, так и в графическом. Рассмотрим **основные** функции видеосервиса BIOS, которые вызываются через прерывание 10h, их номер обычно заносится в регистр AH.

Экраном вывода считается прямоугольная область экрана с базовыми координатами:

<**Левый верхний угол**> ====> CX <CH=x (строка), CL=y (столбец)>

По умолчанию **CX=0**.

<**Правый нижний угол**> ====> DX <DH=x, DL=y>

По умолчанию в DOS устанавливается текстовый режим работы экрана. В этом случае **DX=184Fh** (24 и 79 — весь экран).

Допустимые значения координат <x,y> зависят от режима работы дисплея и типа видеоадаптера — см. табл.14.3 и табл.14.4.

14.4.1. Установка и запрос видеорежима

MOV AH,0 ; Установить видеорежим дисплея

MOV AL, *НомерВидеорежима*

INT 10h

НомерВидеорежима задается в младших 7 битах. Установка в единицу старшего (8) бита означает очистку экрана. Остановимся только на текстовых видеорежимах.

Таблица 14.3. Текстовые видеорежимы.

Регистр AL	Разрешающая способность экрана	Режим и количество поддерживаемых цветов	Номера видеостраниц
0	40 x 25	Черно-белый	0-7
1	40 x 25	Цветной, 16 цветов	0-7
2	80 x 25	Черно-белый	0-3
3	80 x 25	Цветной, 16 цветов	0-3
7	80 x 25	Монохромный	0-3

Для видеоадаптеров, поддерживающих стандарт VESA BIOS Extension, возможно переключение в режимы с высоким разрешением:

MOV AH,4Fh ; Установить расширенный видеорежим дисплея

MOV AL, 02 ; режим SVGA

MOV BX, *НомерВидеорежима*

INT 10h

НомерВидеорежима задается в младших 13 битах. Установка в единицу старшего (15) бита означает, что экран НЕ очищается. Количество поддерживаемых цветов определяется емкостью видеопамяти.

Таблица 14.4. Расширенные текстовые видеорежимы.

Регистр BX	Разрешающая способность экрана
108h	80 x 60
109h	132 x 25
10Ah	132 x 43
10Bh	132 x 50
10Ch	132 x 60



ЗАМЕЧАНИЕ

Будьте осторожны и внимательны, если вы не уверены в поддержке вашим адаптером (видеокартой) стандарта VESA BIOS Extension.

По умолчанию в DOS используется режим 3.

MOV AH,5 ; Установить активную видеостраницу дисплея

MOV AL, *НомерВидеоСтраницы*

INT 10h

Страница, которая в данный момент отображается на экране, называется *активной*. По умолчанию — это страница 0. Можно вывести текст в неактивную страницу, а потом переключиться на нее. Это позволяет повысить скорость вывода — см. п. 14.4.3.

MOV AH,0Fh ; Запрос текущего видеорежима дисплея

INT 10h

На выходе данная функция возвращает регистры:

AL — *НомерВидеорежима*;

AH — число символов в строке;

BH — *НомерВидеоСтраницы*.

14.4.2. Управление размером и положением курсора

Обычно символ курсора похож на знак подчеркивания. Используя прерывание 10h, можно управлять вертикальным размером курсора.

MOV AH,1 ; Установить размер курсора

MOV CH, *ВерхняяЛинияСканирования* ; биты 4-0

MOV CL, *НижняяЛинияСканирования* ; биты 4-0

INT 10h

Курсор сохраняет свой вид, пока программа НЕ изменит его.

MOV AH,2 ; Установить положение курсора
MOV BH, *НомерВидеостраницы*
MOV DH, *НомерСтроки*
MOV DL, *НомерСтолбца*
INT 10h

MOV AH,3 ; Чтение текущего положения и размера курсора
MOV BH, *НомерВидеостраницы*
INT 10h

На выходе данная функция возвращает регистры CX и DX:
DH — номер строки текущей позиции курсора;
DL — номер столбца текущей позиции курсора;
CH, CL — верхняя и нижняя линии сканирования соответственно.

14.4.3. Вывод символов на экран в текстовом режиме

Текстовый режим работы дисплейных адаптеров в IBM-совместимых компьютерах использует часть оперативной памяти для видеопамяти. Адрес ее начала зависит от типа адаптера. Для адаптеров CGA, EGA, VGA и SVGA это B800h:0000. Видеопамять содержит "карту" текстового экрана: коды отображаемых символов и их атрибуты. Прямое обращение к видеопамяти обеспечивает максимально возможную скорость изменения изображения на экране, потому что любой символ, попадающий в видеопамять, отображается на экране немедленно.

Экран в текстовом режиме хранится как последовательность пар байт, где первый байт в каждой паре - любой из 256 символов ASCII , а второй байт - атрибут его вывода.

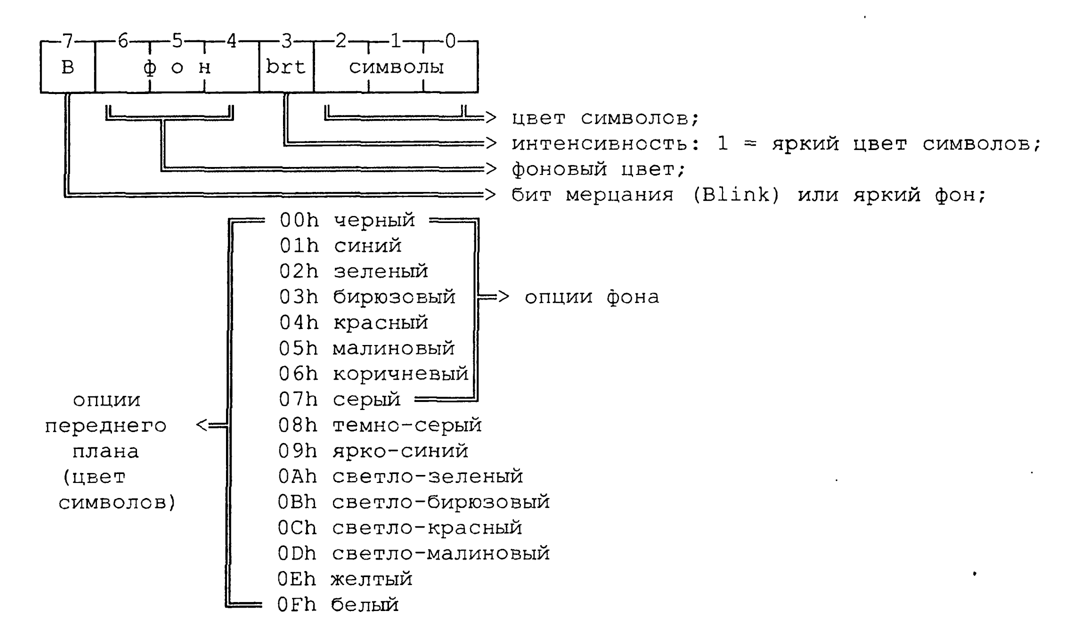


Рис. 14.6. Структура атрибута символа в текстовом режиме работы дисплея.

```
MOV  AH,8  ; Чтение символа и его атрибута в текущей позиции курсора
MOV  BH, НомерВидеостраницы
INT    10h
```

На выходе данная функция возвращает регистры:

AH — атрибут символа;

AL — ASCII-код символа.

```
MOV  AH,9  ; Вывести символ с заданным атрибутом
MOV  BH, НомерВидеостраницы
MOV  AL, ASCII-код_символа
MOV  BL, АтрибутСимвола
MOV  Cx, ЧислоПовторенийСимвола
INT    10h
```

```
MOV  AH,0Ah  ; Вывести символ с текущим атрибутом
MOV  BH, НомерВидеостраницы
MOV  AL, ASCII-код_символа
MOV  Cx, ЧислоПовторенийСимвола
INT    10h
```

С помощью этих двух последних функций можно вывести на экран **любой** символ, включая символы CR и LF.

```
MOV  AH,0Eh  ; Вывести символ с текущим атрибутом в режиме телетайпа
MOV  BH, НомерВидеостраницы
MOV  AL, ASCII-код_символа
INT    10h
```

Эта функция интерпретирует символы CR, LF, BS (08) и BELL (07) как управляющие для форматирования экрана.

14.4.4. Очистка и прокрутка экрана

```
MOV  AH,6  ; Прокрутка экрана вверх
MOV  AL, КоличествоСтрокПрокрутки ; 0 — весь экран заполняется пробелами
MOV  BH, АтрибутЭкрана                ; 7 — черно-белый
MOV  DH, НомерСтрокиНижнегоПравогоУгла ; 24 — весь экран в режимах 0-3,7
MOV  DL, НомерСтолбцаНижнегоПравогоУгла ; 79 — весь экран в режимах 2,3,7
MOV  CH, НомерСтрокиВерхнегоЛевогоУгла
MOV  CL, НомерСтолбцаВерхнегоЛевогоУгла
INT    10h
```

```
MOV  AH,7  ; Прокрутка экрана вниз
MOV  AL, КоличествоСтрокПрокрутки ; 0 — весь экран заполняется пробелами
MOV  BH, АтрибутЭкрана                ; 7 — черно-белый
MOV  DH, НомерСтрокиНижнегоПравогоУгла ; 24 — весь экран в режимах 0-3,7
MOV  DL, НомерСтолбцаНижнегоПравогоУгла ; 79 — весь экран в режимах 2,3,7
MOV  CH, НомерСтрокиВерхнегоЛевогоУгла
MOV  CL, НомерСтолбцаВерхнегоЛевогоУгла
INT    10h
```

14.4.5. Вывод строки символов

MOV AH,13h ; Вывести строку символов
MOV AL, *Режим вывода*
MOV CX, *ЧислоВыводимыхСимволов*
MOV BL, *АтрибутВывода* ; если строка содержит только символы
MOV DH, *НомерСтрокиНачалаВывода*
MOV DL, *НомерСтолбцаНачалаВывода*
LEA ES:BP, *АдресНачала Строки_в_памяти*
INT 10h

Режим вывода задается флажком в нулевом бите:

- 0 — выводится обычная строка (по одному байту на символ). После вывода строки курсор переместить в конец строки.
- 1 — строка наряду с символами содержит и их атрибуты (по два байта на символ).

Эта функция интерпретирует символы CR(0Dh), LF(0Ah), BS (08) и BELL (07) как управляющие.



ЗАМЕЧАНИЕ

Вывод строки в режиме 1 производится значительно быстрее при копировании ее прямо в видеопамять.

ПРИМЕР 14.6

Изменим немного наш пример 14.5, применив наши новые знания по прерыванию 10h видеосервиса BIOS. Сделаем очистку экрана (процедура **ClrScr**), установку курсора в левый верхний угол экрана (процедура **Goto00**) и более яркий (цветной) вывод сообщений об ошибке (процедура **ErrorOut**) с использованием атрибутов вывода символов.

```
;---------- Procedure Goto00 ------------------------
Goto00      Proc      Near
               Push     dx
               Push     ax
               Mov      dh,0    ; Номер Строки
               Mov      dl,0    ; Номер Столбца
         ; Установить положение курсора
               Mov      ah,2
               Mov      bh,0    ; Номер Видеостраницы
               Int      10h
               Pop      ax
               Pop      dx
               Ret
Goto00        EndP
;---------- Procedure ClrScr ------------------------
ClrScr      Proc      Near
               Push     ax
               Push     bx
               Push     cx
               Push     dx
               Mov      ah,6    ; Прокрутка экрана вверх
```

```
; Количество строк прокрутки:
;          0 - весь экран заполняется пробелами
                Mov     al,0
                Mov     cx,0   ; Верхний Левый Угол <0,0>
; Номер строки нижнего правого угла:
;          24 - весь экран в режимах 0-3,7
                Mov     dh,24
; Номер столбца нижнего правого угла:
;          79 - весь экран в режимах 2,3,7
                Mov     dl,79
; Атрибут Экрана: 7 - черно-белый
                Mov     bh,7
                Int     10h
                Pop     dx
                Pop     cx
                Pop     bx
                Pop     ax
                Ret
ClrScr      EndP
;---------- Procedure ErrorOut ---------------------
;  ВХОД:
;          LEA  BP, АдресНачалаСтроки_в_памяти
;          MOV  CX, ЧислоВыводимыхСимволов
ErrorOut     Proc
                MOV   AH,13h    ; Вывести строку символов
                MOV   AL,0      ; Режим вывода - обычный
; Атрибут вывода: светло-красный (0Ch)на бирюзовом (3)
                MOV   BL,3Ch
                MOV   DH,10     ; Номер строки начала вывода
                MOV   DL,5      ; Номер столбца начала вывода
                INT      10h
                Ret
ErrorOut     EndP
```

Приведем фрагменты **измененных участков программы**. Обратите внимание на вывод ошибочных сообщений **Err1, Err2, Err3** (символ окончания строки **$** здесь НЕ требуется!). Файл **RW21_10h.asm** целиком можно посмотреть в прилагаемых к книге материалах.

Фрагменты исходного текста программы RW21_10h.asm

```
..................................................
Data     Segment  Byte  Public
         Buffer   db 30,31 dup (' ')  ; Буфер ввода
         X        dw (?)
         L        dw 6  ; размер буфера
         E        db (?)    ; флаг ошибки
         CR_LF    db 0dh,0ah,'$'
         Meet1    db ' Введите X и нажмите <Enter>.   Отмена - <Ctrl-C>',0dh,0ah,'$'
         Meet     db ' (-32768 < X < 32767) = $'
         Result   db ' Assembler : X = $'
         Repeat   db ' Продолжим ? ("N" - выход):$'
         Err1     db ' Ошибка при вводе - введено слишком много символов!'
         Err2     db ' Введена НЕ цифра!'
         Err3     db ' Нарушен диапазон!'
         GoodBye  db ' Good bye my friend',0dh,0ah,'$'
         Res      db 6 dup (?),'$'; Буфер вывода
```

```
Data      EndS
Code      Segment  Byte  Public
.........................................................
;———— Procedure Main ————————————————
Main      Proc   Near
          Mov    ax,data
          Mov    ds,ax
          Mov    Es,ax  ; ES=DS!!!Для вывода строки INT 10h, AH=13h !!!!
 Again:
          Mov    E,0
          Call   ClrScr
          Call   Goto00
          Lea    dx,CR_LF
          Call   Output
.........................................................
;———— Procedure Ascii_Bin ————————————————
; ВХОД:
;    нач. адрес строки ————> BX
;    размер буфера ввода ———> L
;    фактическое число введенных символов  ———> CX
;     сообщения об ошибке:
;       ERR1 - "Введено слишком много символов!"
;       ERR3 - "Нарушен диапазон!"
; ВЫХОД:
;    число ———> AX и X
;     если ошибок НЕТ, то CF=0, DI=0FFh
;        иначе        CF=1, DI <—— адрес ошиб. символа
;          если введено > L символов, то E=1
;          если нарушен диапазон, то E=3
;————————————————————————————————
Ascii_Bin Proc   Near
          Push   bx
          Push   cx
          Xor    ax,ax
          Mov    di,0ffh              ; ошибок НЕТ
          Cmp    cx,L                 ; проверка количества введенных символов
          Ja     Bad                  ; CX > L
BlanKs:                               ; пропуск ведущих пробелов
          Cmp    Byte  Ptr [bx],' '
          Jne    ChkNeg               ; пробелов НЕТ
          Inc    bx
          Loop   Blanks
ChkNeg:
          Cmp    Byte  ptr [bx],'-'   ; число ОТРИЦАТЕЛЬНОЕ ?
          Jne    ChkPos               ; НЕТ
          Inc    bx                   ; след. символ
          Dec    cx
          Call   Conv_St ; преобразование ОТРИЦАТЕЛЬНОГО числа
          Jc     Quit   ; есть ошибки при преобразовании строки
          Cmp    ax,32768 ; проверка допустимого диапазона
          Ja     BadRange             ; AX > 32768 (WORD!!!)
          Neg    ax                   ; МЕНЯЕМ ЗНАК
          Clc                         ; CF=0 (НЕТ ошибок)
          Js     Good
ChkPos:
```

```
        Cmp     Byte  ptr [bx],'+'      ; есть знак '+'?
        Jne     Go_Conv                 ; НЕТ
        Inc     bx                      ; пропуск знака '+'
        Dec     cx
Go_Conv:
        Call    Conv_St ; преобразование ПОЛОЖИТЕЛЬНОГО числа
        Jc      Quit   ; есть ошибки при преобразовании строки
        Cmp     ax,32767                ; AX > 32767 (WORD!!!)
        Ja      BadRange
Good:                                   ; OK!
        Clc
        Jnc     Quit
Bad:                                    ; CX > L
        LEA     BP, Err1
        MOV     CX, 51
        Call    ErrorOut
        Mov     E,1
        Stc                             ; CF=1
        Jmp     Quit
BadRange:                               ; выход за диапазон
        LEA     BP, Err3
        MOV     CX, 18
        Call    ErrorOut
        Mov     E,3
        Stc                             ; CF=1
Quit:
        Pop     cx
        Pop     bx
        Mov     X,ax
        Ret
Ascii_Bin  Endp

;——————— Procedure Conv_St ———————————————
; посимвольное преобразование строки во внутреннее представление числ
; ВХОД:
;     нач. адрес строки в буфере ввода ————> BX
;     ERR2 - сообщение "НЕ цифра!"
; ВЫХОД:
;     число ———> AX
;        если введена не цифра, то CF=1, DI <—— адрес ошиб. символа,
;                             флаг E=2
;——————————————————————————————————————
Conv_St   Proc    Near
        Push    bp
        Push    bx
        Push    si
        Mov     bp,bx
        Xor     bx,bx
        Mov     si,10
Range:                          ; проверка на допустимый символ [0,...,9]
        Cmp     Byte  ptr ds:[bp],'0'
        Jb      NoneDigit               ; < '0'
        Cmp     Byte  ptr ds:[bp],'9'
        Jbe     Digit                   ; <= '9'
NoneDigit:                              ; НЕ цифра
        LEA     BP, Err2
```

```
        MOV    CX, 18
        Call   ErrorOut
        Mov    E,2
        Mov    di,bp                ; адрес ошиб. символа —> DI
        Stc                         ; CF=1
        Jc     End_Conv
Digit:                              ; обработка цифры
        Push   dx
        Mul    si                   ; <DX:AX> = AX*10
        Pop    dx
        Mov    bl,ds:[bp]           ; ASCII - код
        And    bx,0fh               ; выделение МЛ. части
        Add    ax,bx
        Jc     End_Conv  .          ; если результат велик
        Inc    bp
        Loop   Range
        Clc                         ; CF=0
End_Conv:
        Pop    si
        Pop    bx
        Pop    bp
        Ret
Conv_St  Endp
.........................................................
CODE      EndS
     End      Start
```

14.5. Ввод с клавиатуры. Прерывание BIOS 16h

BIOS предоставляет достаточно много функций для работы с клавиатурой. Причем некоторые функции являются уникальными. Например, поместить символ в буфер клавиатуры, как если бы он был введен пользователем на самом деле (функция 5). Или функция 2 (12h, 22h) проверки состояния клавиатуры. Функции BIOS считывания данных с клавиатуры могут быть рассчитаны и на конкретный тип клавиатуры с определенным числом клавишей. Различают обычную клавиатуру (**83/84-key**) и две расширенные (**101/102-key** и **122-key**)

14.5.1. Чтение символа БЕЗ эхо-сопровождения

```
MOV  AH,0  ; для клавиатуры 83/84-key
INT   16h

MOV  AH,10h  ; для клавиатуры 101/102-key
INT   16h

MOV  AH,20h  ; для клавиатуры 122-key
INT   16h
```

На выходе данная функция возвращает регистры:

AL — *ASCII_Символ*, 0 или префикс scan-кода

AH — *ScanКодФункциональнойКлавиши*

Функциональные клавиши НЕ вырабатывают какого-либо символа. Они используются в основном для выполнения каких-либо действий. Scan-коды функциональных клавиш представлены в табл. П3.1 и табл. П3.2.

Если нажимается какая-либо функциональная клавиша, то в регистре AL будет содержаться НОЛЬ. Поэтому после выполнения данного прерывания нужно обязательно проверять содержимое регистра AL:

```
MOV     AH, 0
INT     16h
CMP     AL, 0
JZ      ScanCode
.......................................................
ScanCode:
; Обработка Scan-кода (содержимое регистра AH)
```

14.5.2. Определение наличия введенного символа

```
MOV     AH,1  ;  для клавиатуры 83/84-key
INT     16h

MOV     AH,11h  ;  для клавиатуры 101/102-key
INT     16h

MOV     AH,21h  ;  для клавиатуры 122-key
INT     16h
```

На выходе данная функция возвращает регистры:

AL — *ASCII_Символ*

AH — *ScanКодФункциональнойКлавиши*

и сбрасывает флаг нуля (ZF=0), если в буфере имеется считанный символ.

ZF=1, если буфер пуст.

14.5.3. Запись символа в буфер клавиатуры

Буфер клавиатуры организован по принципу кольца и имеет длину 16 байт. В нем символ имеет длину 16 бит (scan-код клавиши и собственно символ ASCII)

```
MOV     AH, 5  ;  Поместить символ в буфер клавиатуры
MOV     CH, ScanКодФункциональнойКлавиши
MOV     CL, ASCII_Символ
INT     16h
```

На выходе данная функция возвращает содержимое регистра:

AL — 00, если операция выполнена успешно;

AL — 01h, если буфер клавиатуры переполнен.

14.5.4. Определение текущего состояния клавиатуры

```
MOV     AH,2  ;  для клавиатуры 83/84-key
INT     16h
```

```
MOV   AH,12h  ; для клавиатуры 101/102-key
INT   16h

MOV   AH,22h  ; для клавиатуры 122-key
INT   16h
```

На выходе данная функция возвращает содержимое регистров:
AL — байт состояния клавиатуры 1;
AH — байт состояния клавиатуры 2 (только для функций 12h и 22h).

Байт состояния клавиатуры 1 всегда находится в оперативной памяти по адресу 0000h:0417h или 0040h:0017h:
Бит 7 — клавиша **Ins** включена;
Бит 6 — клавиша **CapsLock** включена;
Бит 5 — клавиша **NumLock** включена;
Бит 4 — клавиша **ScrollLock** включена;
Бит 3 — клавиша **Alt** нажата (только ЛЕВАЯ для функций 12h, 22h);
Бит 2 — клавиша **Ctrl** (любая) нажата;
Бит 1 —ЛЕВАЯ клавиша **Shift** нажата;
Бит 0 —ПРАВАЯ клавиша **Shift** нажата.

Байт состояния клавиатуры 2 всегда находится в оперативной памяти по адресу 0000h:0418h или 0040h:0018h:
Бит 7 — клавиша **SysRq** нажата;
Бит 6 — клавиша **CapsLock** нажата;
Бит 5 — клавиша **NumLock** нажата;
Бит 4 — клавиша **ScrollLock** нажата;
Бит 3 — ПРАВАЯ клавиша **Alt** нажата;
Бит 2 — ПРАВАЯ клавиша **Ctrl** нажата;
Бит 1 —ЛЕВАЯ клавиша **Alt** нажата;
Бит 0 — ЛЕВАЯ клавиша **Ctrl** нажата.

14.6. Введение в программирование на уровне портов ввода-вывода

Системные функции обычно работают с устройствами компьютера напрямую через порты ввода-вывода. Это — самый нижний уровень ввода-вывода, который требует **детального** знания аппаратуры. Чтобы выяснить, какие конкретно порты на вашем компьютере обслуживают вашу аппаратуру, можно сделать следующее: запустить известную программу **SiSoft Sandra** и вызвать модуль **I/O Settings**. В результате будут показаны ВСЕ порты и соответствующие им устройства.

14.6.1. Команды чтения операндов из порта INx

Существует несколько команд чтения информации из порта. Их выполнение НЕ влияет на флаги. Наиболее простая команда **IN** (INput operand from port) служит для чтения из порта **одного** байта, слова или двойного слова:

IN *регистр-аккумулятор, НомерПорта*

Логика работы команды:

Из порта *НомерПорта* передается байт, слово или двойное слово соответственно в *регистр-аккумулятор* **AL**, **AX** или **EAX**. *НомерПорта* можно задать либо непосредственно (если он лежит в диапазоне 0-255), либо через регистр **DX** (если он превышает значение 255).

Следующая группа команд (INput String Byte/Word/Double word operands from port) появилась, начиная с процессора i286, и служит для чтения **строки** байт, слов или двойных слов из порта, номер которого хранится в регистре **DX**:

INSB

INSW

INSD

Эти команды явных операндов НЕ имеют. Адрес строки-приемника должен быть предварительно загружен в регистр **ES:EDI (ES:DI)**. Команды могут иметь префикс **REP**x. Логика работы этих команд всецело определяется логикой работы цепочечных команд(п. 11.1).

14.6.2. Команды записи операндов в порт OUTx

Команды этой группы обратны командам предыдущей группы. Их выполнение тоже НЕ влияет на флаги. Наиболее простая команда **OUT** (OUTput operand to port) служит для записи в порт **одного** байта, слова или двойного слова:

OUT *НомерПорта, регистр-аккумулятор*

Логика работы команды:

Из *регистра-аккумулятора* **AL**, **AX** или **EAX** передается соответственно байт, слово или двойное слово в порт *НомерПорта*. *НомерПорта* можно задать либо непосредственно (если он лежит в диапазоне 0-255), либо через регистр **DX** (если он превышает значение 255). Эта команда служит для прямого управления оборудованием компьютера.

Следующая группа команд (OUTput Byte/Word/Double word String to port) появилась, начиная с процессора i286, и служит для записи **строки** байт, слов или двойных слов в порт, номер которого хранится в регистре **DX**:

OUTSB
OUTSW
OUTSD

Эти команды явных операндов НЕ имеют. Адрес строки-источника должен быть предварительно загружен в регистр **DS:ESI (DS:SI)**. Команды могут иметь префикс **REP**x. Логика работы этих команд всецело определяется логикой работы цепочечных команд — см. п. 11.1.

В качестве простейшего примера работы с портами рассмотрим включение и выключение динамика:

```
; Работа с динамиком IBM PC
IN      AL, 61h    ; текущее состояние порта
OR      AL, 3       ; установить биты 0 и 1
OUT     61h,AL     ; Включить динамик IBM PC
.............................................
IN      AL, 61h    ; текущее состояние порта
AND     AL, 11111100b       ; обнулить биты 0 и 1
OUT     61h,AL     ; Выключить динамик IBM PC
```



ЗАМЕЧАНИЕ.

Иногда при выполнении команд INx/OUTx внешнее устройство не успевает отреагировать, поэтому в некоторых случаях может потребоваться установка задержки.

14.6.3. Контроллер клавиатуры

Контроллеру клавиатуры соответствуют обычно порты с номерами от 60h до 6Fh. Например, на моем компьютере, работающем под управлением Windows 2000 Professional, при запуске программы SiSoft Sandra (модуль I/O Settings) было получено, что клавиатуру Standard 101/102-key or Microsoft Natural PS/2 Keyboard обслуживают порты 60h и 64h. Аналогичную информацию можно получить, запустив следующую цепочку (**Windows 2000 Professional**):

Панель управления | Клавиатура | Оборудование | Свойства | Ресурсы.

Контроллер клавиатуры выполняет следующие действия:

- прием данных от клавиатуры;
- проверка ошибки четности при передаче данных с клавиатуры;
- кодирование полученной информации;
- передача байта во входной буфер клавиатуры;
- передача байта в выходной буфер и извещение процессора.

Опишем несколько характерных особенностей контроллера клавиатуры.

14.6.3.1. Регистр состояния и буферы контроллера клавиатуры

Регистр состояния контроллера клавиатуры доступен по чтению из порта **64h**:

- бит 0 — выходной буфер клавиатуры содержит данные (= 1);
- бит 1 — в буфере ввода есть данные (= 1);
- бит 2 — самотестирование клавиатуры закончено (= 1);
- бит 3 — признак интерпретации полученного байта (бит = 1 — данное, бит = 0 - команда);
- бит 4 — клавиатура заблокирована (= 0);
- бит 5 — передача байта не закончена — тайм-аут при передаче (= 1);
- бит 6 — прием байта от клавиатуры не закончен — тайм-аут при приеме (= 1);
- бит 7 — ошибка четности при передаче данных от клавиатуры (= 1).

Входной буфер контроллера доступен по записи: порт **60h** (данные), порт **64h** (команды).

Выходной буфер контроллера доступен по чтению из порта **60h**.

16.6.3.2. Команды контроллера клавиатуры

Команды передаются через входной буфер клавиатуры - порт **64h.** Если у команды есть параметр, то он должен быть помещен в порт **60h.**

- **Команда 20h** — запись значения управляющего байта в выходной буфер.
- **Команда 60h** — запись значения управляющего байта контроллера. Значение его битов следующее:

   - бит 0 — если выходной буфер содержит данные, то генерируется прерывание от контроллера клавиатуры (= 1);
   - бит 1 — обычно равен 0;
   - бит 2 — заменяет собой соответствующий бит управляющего регистра контроллера;
   - бит 3 — переключатель замка клавиатуры перестает действовать (= 1);
   - бит 4 — клавиатура отключается (= 1);
   - бит 5 — задание режима интерфейса: НЕ выполняется проверка четности и НЕ преобразуются scan-коды (= 1);
   - бит 6 — бит совместимости;
   - бит 7 — равен 0.
- **Команда 0AAh** — внутренний тест контроллера. Если ошибок нет, то в выходном буфере содержится 55h. После выполнения данной команды нужно восстановить значение управляющего байта контроллера (см. команды 20h и 60h).
- **Команда 0ABh** — выполнение теста интерфейса контроллер-клавиатура. Результат помещается в выходной буфер контроллера:
   - 00 — ошибок нет;
   - 01 — сигнал линии часов завис в нижнем положении;
   - 02 — сигнал линии часов завис в верхнем положении;
   - 03 — сигнал линии данных завис в нижнем положении;
   - 04 — сигнал линии данных завис в верхнем положении.
- **Команда 0ACh** — выполнение диагностического дампа памяти контроллера, текущего состояния входного и выходного порта, а также слова состояния контроллера. В конце выдается строка, идентифицирующая фирму изготовитель.
- **Команда 0ADh** — блокирование клавиатуры.
- **Команда 0AEh** — разблокирование клавиатуры.
- **Команда 0C0h** — чтение входного порта.
- **Команда 0D0h** — чтение содержимого байта выходного порта контроллера клавиатуры. Значение его битов следующее:
   - бит 0 — порт подсоединен к линии сброса системы (= 1);
   - бит 1 — состояние дополнительной адресной линии;
   - биты 2 и 3 — значения их не определены;
   - бит 4 — выходной буфер полон (= 1);
   - бит 5 — входной буфер пуст (= 1);
   - бит 6 — определяет значение линии часов при передаче данных в клавиатуру;
   - бит 7 — определяет значение линии данных при их передаче в клавиатуру.
- **Команда 0D1h** — поместить байт в выходной порт контроллера клавиатуры.
- **Команда 0E0h** — поместить информацию о состоянии входных линий из порта состояния входных линий в выходной буфер:
   - 0 — состояние входной линии часов;
   - 1 — состояние входной линии данных.

- **Команды 0F0h-0FFh** — сброс выходного порта.

14.6.3.3. Команды управления клавиатурой

Эти команды передаются через порт **60h** в режиме записи. Если необходим параметр, то он передается ПОСЛЕ подтверждения полученной команды. Ответ посылается на все команды, кроме **0EEh** и **0FEh**.

- **Команда 0EDh** — изменение состояния светодиодов на клавиатуре. Второй байт этой команды задает их новое состояние (1 — включить, 0 - выключить):
  - бит 0 — **Scroll Lock**;
  - бит 1 — **Num Lock**;
  - бит 2 — **Caps Lock.**

При этом состояние переключателей, хранящихся в BIOS, НЕ изменяется. При первой же возможности обработчик прерывания клавиатуры BIOS восстанавливает состояние светодиодов.

- **Команда 0EEh** — эхо-запрос клавиатуры, на который клавиатура отвечает тем же scan-кодом **0EEh** (используется для диагностики).
- **Команда 0F0h** — установка или запрос таблицы scan-кодов, определяемый параметром:
  - 0 — получить номер таблицы;
  - 1 — установить таблицу 1;
  - 2 — установить таблицу 2;
  - 3 — установить таблицу 3.
- **Команда 0F3h** — установка параметров режима автоповтора:
  - **Пауза перед началом автоповтора** (в миллисекундах) определяется следующей формулой: **(1 + 2*b6 + b5)*250** и задается двоичным набором бит 6-5 параметра команды:

    00b — 250ms;
    01b — 500ms;
    10b — 750ms;
    11b — 1000ms.
  - **Скорость автоповтора** (символов в секунду) определяется следующей формулой: **4.17*(8 + 4*b2 + 2*b1 + b0)*2*(2*b4 + b3)** и задается двоичным набором бит 4-0 параметра команды:

    00000b — 30,0;
    00010b — 24,0;
    00100b — 20,0;
    00111b — 16,0;
    01000b — 15,0;
    01010b — 12,0;
    01100b — 10,0;
    01111b — 8,0;
    10010b — 6,0;

10100b — 5,0;
10111b — 4,0;
11010b — 3,0;
11111b — 2,0.

- **Команда 0F4h** — включение клавиатуры.
- **Команда 0F5h** — установка всех характеристик клавиатуры в исходное состояние (по умолчанию) и блокирование клавиатуры.
- **Команда 0F6h** — аналогична предыдущей команде, но клавиатура не блокируется.
- **Команды 0F7h-0FAh** — задание характера реакции клавиш:
  - **0F7h** — все клавиши повторяемые;
  - **0F8h** — все клавиши посылают код нажатия и прерывания (клавишу отпустили);
  - **0F9h** — все клавиши посылают только код нажатия;
  - **0FAh** — все клавиши повторяемые и посылают код нажатия и прерывания.
- **Команды 0FBh-0FDh** — задание действия отдельных клавиш:
  - **0FBh** — клавиша повторяемая;
  - **0FCh** — клавиша посылает код нажатия и прерывания (клавишу отпустили);
  - **0FDh** — клавиша посылает только код нажатия;
- **Команда 0FEh** — повторить передачу.
- **Команды 0FFh** — выполнить самотестирование и сброс клавиатуры. Команда возвращает байт **0AAh**, если все удачно, и **0FCh** в случае ошибки.

Клавиатура отвечает на все команды, кроме **0EEh**, **0FEh** и **0FFh**, scan-кодом **0FAh** (подтверждение), который поглощается стандартным обработчиком BIOS.

Пример тестирования клавиатуры см. в конце главы 15.



Макросредства языка Ассемблера IBM PC

Каждому моменту на практике предшествует альтернативное решение.

Лукач Дьердь (1885-1971 гг.)

Макросредства языка Ассемблера IBM PC являются одним из самых мощных средств языка Ассемблера, который еще называют МАКРОАССЕМБЛЕРОМ за способность оперировать с набором команд как с одной командой. Мы с вами, уважаемый читатель, уже достаточно освоились с основными элементами Ассемблера, разработали много программ и процедур. Теперь самое время этот наш опыт обобщить и сохранить некоторые процедуры несколько в другом виде — в виде макросов.

Использование макросредств позволяет:

- сделать программу более понятной — за счет применения макрокоманд с параметрами;
- упростить и сократить исходный текст программы;
- уменьшить число возможных ошибок кодирования — за счет использования уже отлаженных макрокоманд;
- динамически изменять программу за счет изменения входных параметров;
- использовать директивы условного ассемблирования и управления листингом;
- сделать свою библиотеку макроопределений;
- существенно ускорить выполнение программы за счет отсутствия накладных расходов на вызов процедур (см. п. 9.1.2) — правда, при этом может возрасти длина программы;
- воспользоваться уже разработанными библиотеками макроопределений (часто они снабжаются расширением имени файла **inc** — include) и вставлять их в свою программу, что позволяет ускорить программирование на Ассемблере.

Использование макросов имеет и свои недостатки:

- макросы хранятся в виде исходных файлов и при подключении к программе нуждаются в ассемблировании, и, если их много, то может существенно увеличиться время ассемблирования;
- макросы могут легко скрывать результаты работы команд.

С макросами широко работает язык С на уровне описания функций с помощью препроцессорной директивы **#define** и язык С++ — на уровне **inline**-функций. Но эти языки накладывают на макросы достаточно существенные ограничения, чего, конечно, нет в Ассемблере.

15.1. Основные понятия. Макроопределение и макрокоманда

Макросредства языка Ассемблера IBM PC включают в себя три составляющие:

1. **Макроопределение (макрос).** Это — набор команд, содержащий описание какого-то действия или алгоритма. Любую ассемблерную процедуру в принципе можно сделать макросом. *Макроопределение должно находиться в самом начале программы, до определения сегментов.* Макрос имеет следующую структуру:

```
ИмяМакроса   MACRO  [ФормальныеПараметры]
;;  тело макроса
                    EndM
```

2. **Макрокоманда** — краткая ссылка на макроопределение (вызов макроса):

```
ИмяМакроса    [ФактическиеПараметры]
```

3. **Макрорасширение (макроподстановка, макровставка)** — вставка вместо макрокоманды макроса с заменой формальных параметров фактическими, если они есть.

Макроопределение может быть простое и вложенное, т.е. содержать в себе другое макроопределение. Уровень вложенности макроопределений может быть любым. Из одного макроса можно вызывать другие макросы. Команды в макросе могут быть любые.



ОБРАТИТЕ ВНИМАНИЕ

на наличие ДВУХ символов ТОЧКА С ЗАПЯТОЙ (;;) в комментариях — это *макрооператор* и означает признак подавления вывода комментария в листинг макрорасширения.



ПРИМЕР 15.1

Написать программу **целочисленного знакового деления** для любых 16-разрядных данных: X = a/b. Операцию деления и вывода строки сделать в виде макроса.

В качестве основы для решения задачи воспользуемся примером 14.5. Процедура **Main** практически вся будет изменена, хотя логика (алгоритм ее работы) останется прежней. Остальные участки программы, кроме процедур **ReadKeys** и **Bin_Ascii**, тоже немного изменятся. Выделим **жирным шрифтом** новые или измененные участки нашей программы, а *макросредства отметим курсивом*. Макроопределения для операции деления и вывода строки сделаем с формальными параметрами. В данном случае, если параметров несколько, они записываются через запятую.

Фрагменты исходного кода программы IdivW.asm

```
    Title   R_Wnumbr.asm
;CopyRight by Голубь Н.Г., 1993, 2001
;─────────────────────────────
;          МАКРООПРЕДЕЛЕНИЯ
;─────────────────────────────
IdivW        MACRO      y, a, b ;; y=a/b
               Push       ax bx dx
               Mov        ax,a
               Mov        bx,b
               CWD
               IDIV       bx
               Mov        y, ax
               Pop        dx bx ax
               EndM
MessageOut MACRO     StringName
;; Изобразить строку с сообщением DS:StringName
;; Признак окончания строки '$'
               Push       dx
               Lea        DX,DS:StringName
               MessageDX
               Pop        dx
               EndM
MessageDX   MACRO    ;; аналог Procedure Output
;; Вывод строки символов
               Push     ax
               Mov      ah,9
               Int      21h
               Pop      ax
               EndM
SStack       Segment    Byte    Public
               db     64 dup   ('stack')
SStack       Ends
Data          Segment    Byte    Public
Buffer        db 30,31 dup (' ')   ; Буфер ввода
X             dw ?
a             dw ?
b             dw ?
L             dw 6   ; размер буфера
E             db (?)      ; флаг ошибки
CR_LF         db 0dh,0ah,'$'
MeetA         db ' Введите a и нажмите <Enter>',0dh,0ah,'$'
MeetB         db ' Введите b и нажмите <Enter>',0dh,0ah,'$'
ZeroB         db ' Значение b = 0. Деление на НОЛЬ НЕВОЗМОЖНО!!! '
              db 'Повторите ввод ',0dh,0ah,'$'
ValRange      db ' [-32768..32767]=====> $'
Result        db ' X = a/b = $'
DivOverflow   db 'Переполнение при ДЕЛЕНИИ!!!!!!$'
Repeat        db ' Продолжим ? ("N" — выход):$'
Err1          db 0dh,0ah
       db ' Ошибка при вводе — введено слишком много символов!'
       db ' ПОВТОРИТЕ ВВОД!!!',0dh,0ah,'$'
Err2   db 0dh,0ah,' Введена НЕ цифра! ПОВТОРИТЕ ВВОД!!!'
       Db 0dh,0ah,'$'
Err3   db 0dh,0ah,' Нарушен диапазон! ПОВТОРИТЕ ВВОД!!!'
```

```
            Db 0dh,0ah,'$'
GoodBye   db ' Good bye my friend',0dh,0ah,'$'
Res        db 6 dup (?),'$'; Буфер вывода
Data         EndS
Code         Segment    Byte    Public
               Assume   cs:code,ds:data,ss:SStack
;------------------------------
Start:
               Call     Main
               Mov      ax,4c00h
               Int      21h
;----- Procedure Main ---------------
Main         Proc      Near
               Mov      ax,data
               Mov      ds,ax
 Again:
               Mov      E,0
     ; Ввод значения a
               MessageOut MeetA
               MessageOut ValRange
               Call     ReadKeys
               Mov      bx,dx
               Call     Ascii_Bin
               Cmp      E,0
               JZ       NEXT   ; Ошибок НЕТ
               JMP      Again
NEXT:        Mov      a,ax
               MessageOut CR_LF
     ; Ввод значения b
AgainB:
               Mov      E,0
               MessageOut MeetB
               MessageOut ValRange
               Call     ReadKeys
               Mov      bx,dx
               Call     Ascii_Bin
               Cmp      E,0
               Jg       AgainB
               Mov      b,ax
               CMP      ax,0      ; b=0?
               JNZ      NoZERO   ; НЕТ
               jmp      ZERO      ; ДА
     ; X = a/b
NoZERO:
               CMP      ax,-1
               JNZ      CONT
               CMP      a,-32768
               JNZ      CONT
               JMP      DivOver
CONT:
               IdivW    X, a, b
               MessageOut CR_LF
               MessageOut CR_LF
     ; Вывод результата
               MessageOut Result
```

```
            Lea       bx,Res
            Mov       ax,X
            Call      Bin_Ascii
            MessageOut [bx]
            ;Lea       dx,[bx]
            ;Call      Output
 Erro:
            MessageOut CR_LF
            MessageOut CR_LF
      ; продолжать ли решение задачи?
            MessageOut Repeat
            Mov       ah,1
            Int       21h
            Cmp       al,'n'
            Je        Bye
            Cmp       al,'N'
            Je        Bye
            Jmp       Again
 Bye:
            MessageOut CR_LF
            MessageOut GoodBye
            Ret
ZERO:   ; ДЕЛЕНИЕ НА НОЛЬ!!!!!!!!!!
            MessageOut CR_LF
            MessageOut ZeroB
            JMP       AgainB
DivOver: ; Переполнение при ДЕЛЕНИИ: -32768/(-1)=32768
            MessageOut CR_LF
            MessageOut DivOverflow
            JMP        Erro
Main         Endp
;——— Procedure ReadKeys ——————————
ReadKeys    Proc       Near
.........................................................
ReadKeys    Endp
;——— Procedure Ascii_Bin ——————————
Ascii_Bin   Proc       Near
.........................................................
 Bad:                              ; CX > L
            MessageOut Err1   ; Введено слишком много символов
            Mov       E,1
            Stc                                     ; CF=1
            Jmp       Quit
 BadRange:                        ; выход за диапазон
            MessageOut Err3
            Mov       E,3
            Stc                                     ; CF=1
 Quit:
            Pop       cx
            Pop       bx
            Mov       X,ax
            Ret
Ascii_Bin   Endp
;——— Procedure Conv_St ——————————
Conv_St     Proc       Near
```

```
..........................................................
NoneDigit:                              ; НЕ цифра
            MessageOut Err2
            Mov     E,2
            Mov     di,bp               ; адрес ошиб. символа -> DI
            Stc                         ; CF=1
            Jc      End_Conv
 Digit:                                 ; обработка цифры
            Push    dx
            Mul     si                  ; <DX:AX> = AX*10
            Pop     dx
            Mov     bl,ds:[bp]          ; ASCII — код
            And     bx,0fh              ; выделение МЛ. части
            Add     ax,bx
            Jc      End_Conv            ; если результат велик
            Inc     bp
            Loop    Range
            Clc                         ; CF=0
 End_Conv:
            Pop     si
            Pop     bx
            Pop     bp
            Ret
Conv_St     Endp
;——— Procedure Bin_Ascii ———————
Bin_Ascii   Proc    Near
..........................................................
Bin_Ascii   Endp
CODE        EndS
            End     Start
```

Тестовые примеры:

```
 Введите a и нажмите <Enter>
 [-32768..32767]=====> -1111111
  Ошибка при вводе — введено слишком много символов! ПОВТОРИТЕ
  ВВОД!!!
 Введите a и нажмите <Enter>
 [-32768..32767]=====> 111111
 Нарушен диапазон! ПОВТОРИТЕ ВВОД!!!
 Введите a и нажмите <Enter>
 [-32768..32767]=====> 11111
 Введите b и нажмите <Enter>
 [-32768..32767]=====> 11h
 Введена НЕ цифра! ПОВТОРИТЕ ВВОД!!!
 Введите b и нажмите <Enter>
 [-32768..32767]=====> 11
 X = a/b = 1010
 Введите a и нажмите <Enter>
 [-32768..32767]=====> -32768
 Введите b и нажмите <Enter>
 [-32768..32767]=====> -1
Переполнение при ДЕЛЕНИИ!!!!!!
 Введите a и нажмите <Enter>
```

```
 [-32768..32767]=====>  -32768
Введите b и нажмите <Enter>
[-32768..32767]=====> 1
X = a/b = -32768

Продолжим ? ("N" - выход):n
Good bye my friend
```

При внимательном анализе исходного файла мы можем увидеть несколько странных (избыточных), на первый взгляд, передач управления типа такой:

```
                CMP      ax,0      ; b=0?
                JNZ      NoZERO    ; НЕТ
                jmp      ZERO      ; ДА
           ; X = a/b
NoZERO:
                CMP      ax,-1
                JNZ      CONT
                CMP      a,-32768
                JNZ      CONT
                JMP      DivOver
CONT:
                IdivW    X, a, b
                MessageOut CR_LF
```

Почему нельзя было написать проще, так, как мы и писали до сих пор? Например, так:

```
                CMP      ax,0      ; b=0?
                jZ       ZERO      ; ДА
           ; X = a/b
                CMP      ax,-1
                JNZ      CONT
                CMP      a,-32768
                JZ       DivOver   ; Переход, если a=-32768 и b=-1
CONT:
                IdivW    X, a, b
                MessageOut CR_LF
```

Чтобы ответить на этот вопрос, надо понять, что такое макрорасширение (макроподстановка).

15.2. Макрорасширение

Увидеть **полное макрорасширение** можно в файле листинга **Idiww.lst** (ключ **/la** для **tasm**). Сам этот файл получился достаточно большим — вы можете посмотреть его целиком в прилагаемых к книге материалах, а лучше — сделать самим. Посмотрим внимательно на привлекший наше внимание фрагмент кода в файле листинга:

```
185     0074   3D 0000                  CMP    ax,0  ; b=0?
186     0077   75 03                    JNZ    NoZERO     ; НЕТ
187     0079   E9 009E                  jmp    ZERO  ; ДА
188                                  ; X =  a/b
189     007C                         NoZERO:
190     007C   3D FFFF                  CMP    ax,-1
191     007F   75 0B                    JNZ    CONT
192     0081   81 3E 0022r 8000         CMP    a,-32768
```

```
      193   0087   75 03              JNZ    CONT
      194   0089   E9 00A7            JMP    DivOver
      195   008C              CONT:
      196                           IdivW  X, a, b
1     197   008C   50 53 52           Push   ax bx  dx
1     198   008F   A1 0022r           Mov    ax,a
1     199   0092   8B 1E 0024r        Mov    bx,b
1     200   0096   99                 CWD
1     201   0097   F7 FB              IDIV   bx
1     202   0099   A3 0020r           Mov    X, ax
1     203   009C   5A 5B 58           Pop    dx bx  ax
      204                           MessageOut CR_LF
1     205   009F   52                 Push   dx
1     206   00A0   BA 0029r           Lea    DX,DS:CR_LF
1     207                           MessageDX
2     208   00A3   50                 Push ax
2     209   00A4   B4 09              Mov  ah,9
2     210   00A6   CD 21              Int  21h
2     211   00A8   58                 Pop  ax
1     212   00A9   5A                 Pop    dx
```

Он содержит две макрокоманды с фактическими параметрами *IdivW X, a, b* (строка 196) и *MessageOut CR_LF* (строка 204), из которой вызывается третья макрокоманда без параметров *MessageDX* (строка 207). Сразу же за ними появляется **макрорасширение (макровставка)** — выделено жирным шрифтом. Кроме того, в файле листинга номерами показан уровень вложенности макросов — очень удобно!

Таким образом, *если макрокоманд много, то длина кода увеличивается.* А мы знаем, что в условиях 16-разрядного программирования (см. п. 9.2) команды условной передачи управления могут прыгать в пределах байта [-128,...,127]. Поэтому, чтобы все-таки прыгнуть куда надо, пришлось использовать сочетание команд JNZ и JMP — это достаточно распространенный прием в реальном профессиональном программировании.

15.3. Директивы макроассемблера

Директивы позволяют сделать макросы более гибкими. Их достаточно много — рассмотрим основные из них.

15.3.1. Директивы управления листингом

Эти директивы позволяют достаточно гибко управлять областью видимости макрорасширений в файле листинга. Они сохраняют свое действие по всему тексту программы, пока другая директива листинга не изменит его. Эти директивы размещаются в программе так, как нужно программисту.

Перечислим основные директивы управления листингом, как обычно, в формате MASM — соответствие их формату TASM см. в прил. 5.

- .LALL (List ALL) — полный листинг всех макрорасширений. Соответствует в TASM уже известному нам ключу **/la**.
- .LIST (.XALL) — листинг только тех строк макрорасширений, которые порождают машинный код. **Этот режим принят в Ассемблере по умолчанию.**
- .SALL (Suppress ALL) — подавить листинг всех макрорасширений.

- **.XLIST** — подавить (отменить) весь листинг.



ЗАМЕЧАНИЕ

Директивы управления листингом имеют смысл при использовании компилятора TASM с ключом /l или компилятора MASM: **ml /Fl**.

15.3.2. Директива LOCAL

Похожей директивой LOCALS мы уже пользовались — см., например, п. 9.3. Директива **LOCAL** позволяет нам использовать в макросах циклы, операторы перехода, переключатели, т.е. все те конструкции, которые в своем теле содержат *метку*. В противном случае будет выдано сообщение об ошибке Symbol is Multi-Defined (Символ определен многократно). **Эта директива должна быть ПЕРВОЙ в макросе**. Тогда при макрорасширении компилятор будет порождать свои внутренние имена для меток, и дублирование меток будет исключено. Например,

```
;; организация паузы
;; Count — счетчик
Pause           MACRO       Count
                LOCAL       Next
                PUSH        CX
                MOV         CX, Count
Next:
                LOOP        Next
                POP         CX
                ENDM
```



ЗАМЕЧАНИЕ

Компиляторы C++ пока не умеют делать **inline**-функции для подобных конструкций. И теперь вы понимаете почему...

15.3.3. Основные макрооператоры

Макрооператор & (амперсант) позволяет соединить заданные символы с параметром (сделать конкатенацию). Например, обращение к макросу:

```
MoveS   MACRO      tag, Count
        Mov        Cx,Count
REP     MOVS&tag
        ENDM
................................................
        MoveS      w,6
```

породит следующее макрорасширение:

```
        Mov        Cx,6
REP     MOVSw
```

Макрооператор <> (угловые скобки) позволяет передавать группу параметров в виде списка — см. пример 15.2-2 (п.15.3.4).

Макрооператор % (процент) указывает, что находящийся за ним текст является выражением и должен быть предварительно вычислен.

15.3.4. Директивы повторения REPT, IRP и IRPC

Директивы повторения представляют собой макросы Ассемблера (что четко видно в листингах программ) и заставляют компилятор повторять при макрорасширении генерируемую последовательность команд, символов, строк, чисел в соответствии со счетчиком или списком значений. Познакомимся с этими директивами и продемонстрируем их работу на конкретных примерах.

Простейший блок повторений со счетчиком **REPT** имеет следующий формат:

```
         REPT  КоличествоПовторений
;;       тело  макроса
         EndM
```



Создадим макрос **Allocate** для распределения памяти под таблицу (массив байт) определенной длины и занесем в нее значения, начиная с какого-то начального значения с определенным шагом. Сделаем скелет *тестовой* программы и файл листинга, чтобы показать, что порождает при этом макрокоманда — **машинный код** макрорасширений выделен жирным шрифтом.

Исходный текст программы RepeatM.asm

```
          TITLE RepeatM.asm
;CopyRight by Голубь Н.Г., 1993, 2001
;------------------------------
;         МАКРООПРЕДЕЛЕНИЕ
;------------------------------
Allocate     Macro  Table, Length, Value0, Step
;; Распределить память под массив байт Table длиной Length
;; и занести в него последовательные значения с шагом Step,
;;     начиная с Value0
Table          Label  Byte
Val = Value0
  Rept     Length
      DB     Val
      Val  = Val+Step
  EndM
               EndM
SStack       Segment   Byte    Public
               db     64 dup  ('stack')
SStack       Ends
Data          Segment   Byte    Public
  Allocate      Table1, 10, 5, 10   ; макрокоманда
Data          EndS
Code          Segment   Byte    Public
               Assume   cs:code,ds:data,ss:SStack
;---------------------------------
Start:
               Mov      ax,4c00h
               Int      21h
CODE          EndS
              End      Start
```

Фрагмент файла листинга Repeatm.lst

```
Turbo Assembler     Version 4.1     24/11/01   22:50:38       Page 1
RepeatM.asm
.............................................
        20      0000     Data     Segment       Byte      Public
        21               Allocate    Table1, 10, 5, 10   ; макрокоманда

1       22      0000                      Table1      Label    Byte
1       23               =0005                 Val  = 5
1       24                                   Rept      10
1       25                                  DB Val
1       26                                  Val  =   Val+10
1       27                                   EndM
2       28      0000     05                 DB Val
2       29               =000F                   Val  =   Val+10
2       30      0001     0F                 DB Val
2       31               =0019                   Val  =   Val+10
2       32      0002     19                 DB Val
2       33               =0023                   Val  =   Val+10
2       34      0003     23                 DB Val
2       35               =002D                   Val  =   Val+10
2       36      0004     2D                 DB Val
2       37               =0037                   Val  =   Val+10
2       38      0005     37                 DB Val
2       39               =0041                   Val  =   Val+10
2       40      0006     41                 DB Val
2       41               =004B                   Val  =   Val+10
2       42      0007     4B                 DB Val
2       43               =0055                   Val  =   Val+10
2       44      0008     55                 DB Val
2       45               =005F                   Val  =   Val+10
2       46      0009     5F                 DB Val
2       47               =0069                   Val  =   Val+10
        48      000A                      DataEndS
.............................................
```

Блок *неопределенных* повторений с параметром **IRP** имеет следующий формат:

```
            IRP   Параметр, СписокЗначений
      ;;    тело  макроса
            EndM
```

Фактический параметр *СписокЗначений* передается макрокоманде в **угловых скобках** < >.



ПРИМЕР 15.2-2

Создадим макрос **Cube**, который формирует таблицу (массив двойных слов), состоящую из кубов целых чисел. Сделаем скелет *тестовой* программы и файл листинга, чтобы показать, что порождает при этом макрокоманда — **машинный код** макрорасширений выделен жирным шрифтом.

Исходный текст программы IrpM.asm

```
            TITLE IRpM.asm
;CopyRight by Голубь Н.Г., 1993, 2001
;————————————————————————
;           МАКРООПРЕДЕЛЕНИЕ
;————————————————————————
Cube     Macro  Value, IntList
;; Создать таблицу, состоящую из кубов
;; ЦЕЛЫХ чисел, заданных в списке IntList
   IRP     Value, IntList
       DD   Value*Value*Value
   EndM
                 EndM
SStack       Segment    Byte    Public
                 db      64 dup  ('stack')
SStack       Ends
Data          Segment    Byte    Public
Tabl         LABEL   DWORD
     Cube     v, <10,15,13,-111,22,-55>   ; макрокоманда
Data          EndS
Code          Segment    Byte    Public
                Assume   cs:code,ds:data,ss:SStack
;——————————————————————————
Start:
                Mov       ax,4c00h
                Int       21h
CODE          EndS
               End       Start
```

Фрагмент файла листинга Irpm.lst

```
Turbo Assembler    Version 4.1       24/11/01 23:38:48
       Page 1
IrpM.asm
.............................................
      16     0000                    Data Segment      Byte     Public
      17     0000                    Tabl        LABEL   DWORD
      18     Cube v, <10,15,13,-111,22,-55>  ; макрокоманда
1     19                            RP      v, 10,15,13,-111,22,-55
1     20                            DD   v*v*v
1     21                            EndM
2     22     0000   000003E8                DD   10*10*10
2     23     0004   00000D2F                DD   15*15*15
2     24     0008   00000895                DD   13*13*13
2     25     000C   FFEB21B1                DD   -111*-111*-111
2     26     0010   00002998                DD   22*22*22
2     27     0014   FFFD7619                DD   -55*-55*-55
      28     0018                   Data  EndS
.............................................
```

Другой блок *неопределенных* повторений с параметром **IRPC** имеет следующий формат:

```
            IRPC   Параметр, СтрокаЗначений
       ;;   тело макроса
            EndM
```

ПРИМЕР 15.2-3.

Создадим макрос **Characters**, который тоже формирует таблицу (массив слов), состоящую из **целых однозначных чисел**, заданных строкой. Сделаем скелет *тестовой* программы и файл листинга, чтобы показать, что порождает при этом макрокоманда — **машинный код** макрорасширений выделен жирным шрифтом.

Исходный текст программы IrpcM.asm

```
            TITLE IrpcM.asm
;CopyRight by Голубь Н.Г., 1993, 2001
;——————————————————————————
;         МАКРООПРЕДЕЛЕНИЕ
;——————————————————————————
Characters    Macro      Value, CharList
;; Создать таблицу, состоящую из
;; ЦЕЛЫХ чисел, заданных в списке CharList
   IRPC       Value, CharList
   DW         Value
   EndM
              EndM
SStack        Segment    Byte       Public
              db         64 dup     ('stack')
SStack        Ends
Data          Segment    Byte       Public
TablI         LABEL      Word
     Characters    v, 1015131112255   ; макрокоманда
Data          EndS
Code          Segment    Byte       Public
              Assume     cs:code,ds:data,ss:SStack
;——————————————————————————————
Start:
              Mov        ax,4c00h
              Int        21h
CODE          EndS
              End        Start
```

Фрагмент файла листинга Irpcm.lst

```
Turbo Assembler    Version 4.1         24/11/01 23:38:33
        Page 1
IrpcM.asm
.................................................
     16    0000                     DataSegment     Byte     Public
     17    0000                     TablI    LABEL  Word
     18          Characters    v, 1015131112255 ; макрокоманда
1    19                             IRPC     v, 1015131112255
1    20                             DW    v
1    21                                EndM
2    22    0000  0001               DW    1
2    23    0002  0000               DW    0
2    24    0004  0001               DW    1
2    25    0006  0005               DW    5
2    26    0008  0001               DW    1
2    27    000A  0003               DW    3
2    28    000C  0001               DW    1
```

```
2      29     000E   0001          DW    1
2      30     0010   0001          DW    1
2      31     0012   0002          DW    2
2      32     0014   0002          DW    2
2      33     0016   0005          DW    5
2      34     0018   0005          DW    5
       35     001A                  Data EndS
.....................................................
```

15.3.5. Директива выхода EXITM

Эта директива обеспечивает досрочный выход из макроопределения или блока повторения. Кроме того, эта директива может использоваться для выхода из условных блоков.

15.3.6. Директивы условного ассемблирования

Эти директивы позволяют сделать макрос ОЧЕНЬ гибким, поскольку образуют встроенный в Ассемблер мини-язык, похожий на алгоритмические языки. Перечислим основные из этих директив и их формат:

```
IF1
    ;; ассемблировать на первом проходе Ассемблера
ENDIF

IF2
    ;; ассемблировать на втором проходе Ассемблера
ENDIF

IF выражение
    ;; ассемблировать, если выражение истинно
ENDIF

IF выражение
    ;; ассемблировать, если выражение истинно
ELSE
    ;; ассемблировать, если выражение ложно
ENDIF

IF выражение1
    ;; ассемблировать, если выражение1 истинно
ELSEIF выражение2
    ;; ассемблировать, если выражение2 истинно
ELSEIF выражение3
    ;; ассемблировать, если выражение3 истинно
.............................................................
ELSE
    ;; ассемблировать в противном случае
ENDIF
```

```
IFB аргумент
    ;; ассемблировать, если аргумент ПРОПУЩЕН
ENDIF

IFNB аргумент
    ;; ассемблировать, если аргумент НЕ пропущен
ENDIF

IFDEF аргумент
    ;; ассемблировать, если аргумент ОПРЕДЕЛЕН
ENDIF

IFNDEF аргумент
    ;; ассемблировать, если аргумент НЕ определен
ENDIF
```

ПРИМЕР 15.3-1.

Вывести на экран по диагонали символ номер 2 (рожица). Для вывода символа написать макрос.

Исходный текст программы MACROfff.asm

```
                        title    По диагонали рожица
; по мотивам книги Л. Скэнлона, с. 254-255 & П. Абеля, гл.9
;------------------------------
;         МАКРООПРЕДЕЛЕНИЯ
;------------------------------
push_regs           macro    reg_list
;; Сохранение значений регистров в стеке
  IFNB <reg_list>    ;;если список reg_list НЕ пуст
    irp reg,<reg_list>
         push reg
    endm
  ENDIF
                        endm    ;;push_regs
pop_regs            macro    reg_list
;; Восстановление значений регистров из стека
  IFNB <reg_list>
    irp reg,<reg_list>
         pop reg
    endm
  ENDIF
                        endm     ;;pop_regs
move_cursor         macro
;; Перемещение курсора:
;; номер строки ——> регистр DH
;; номер столбца ——> регистр DL
                        mov      ah,2      ;;функция установки позиции
  курсора
                        int      10h
                        endm
```

```
print_AL          macro
;; Изобразить символ, значение которого загружено в AL
                    mov      cx,1    ;;изобразить один символ
                    mov      bl,04Ah ;;атрибут вывода:
           ;;(светло-зеленый <0Ah> символ на красном <4> фоне)
           ;;функция вывода символа на экран в текущей позиции
   курсора
                    mov      ah,9
                    int      10h
                    endm
;*************************************************************
;                            ПРОГРАММА
;*************************************************************
sstack    segment para stack 'stack'
          db       64 dup ('stack')
sstack    ends
code     segment para 'code'
diag_line proc  far
          assume   cs:code,ss:sstack
;организовать нормальный возврат в DOS
          push     ds
          mov      di,0
          push     di
;сохранить регистры
          push_regs <ax,bx,cx,dx>
          sti                           ;разрешить обработку прерываний
;получить номер активной страницы—>BH
          mov      ah,15
          int      10h
          mov      ah,0                 ;задать режим изображения
          mov      al,3                 ;цветной текстовый режим,
   80*25
          int      10h
          mov      dx,0                  ;строка и столбец 0
set:     move_cursor                     ;переместить курсор
          mov      al,2                  ;"рожица"
          print_AL
          inc      dh                    ; по строке
          inc      dl                    ; по столбцу
          inc      dl
          inc      dl
          cmp      dh,25
          jne      set
; Ждать нажатия любой клавиши
          Mov      ah,1
          Int      21h
; Восстановить регистры
          pop_regs <dx,cx,bx,ax>
          ret
diag_line endp
code     ends
          end      diag_line
```

15.3.7. Многострочные комментарии

Мы знаем, что кроме обычных комментариев (символ ; — точка с запятой), принятых в Ассемблере, в макросах используется макрооператор ;; (две точки с запятой), чтобы подавить в макрорасширениях выдачу комментариев. Возможны еще и многострочные комментарии вида:

```
COMMENT  СимволКонцаКомментария
         Любой нужный текст
СимволКонцаКомментария
```

Обычно в качестве *СимволаКонцаКомментария* принимают символ @, если в тексте комментария нет такого символа, или любой другой символ.

15.3.8. Директивы подключения файлов

- **INCLUDE** *имяФайла* — подключение файлов, содержащих макросы, константы, структуры данных. Эта директива аналогична препроцессорной директиве **#include** языка C/C++. Например, следующие файлы входят в поставку MASM-6.12:

```
INCLUDE Dos.inc
INCLUDE Macros.inc
INCLUDE Bios.inc
```

Этими файлами можно воспользоваться для организации в своей программе профессионального ввода-вывода в MS DOS, а также ОЧЕНЬ полезно изучить их для понимания и углубления знаний о прерываниях и макросах.

А подключение файла **win32.inc** (этот файл входит в состав поставки TASM-5) позволит программировать ввод-вывод, используя операционную систему Windows — подробнее см. главу 16 и лабораторную работу № 10.

Подключать можно, конечно, и свои собственные макросы.

- **INCLUDELIB** *имяФайла*— подключение файлов, содержащих библиотечные файлы (с расширением **OBJ** или **LIB**). Например,

```
INCLUDELIB import32.lib
```

Эта библиотека нужна для Windows-программирования.

15.4. Создание библиотеки макросов

Любой алгоритмический язык имеет свои библиотеки, основное назначение которых передать накопленный опыт программирования при реализации тех или иных алгоритмов. Язык Ассемблера тоже их имеет (см. п. 15.3.8). Кроме того, он позволяет программисту создавать и свои библиотеки на исходном языке — библиотеки макросов. Чтобы эти библиотеки служили не только своим создателям, но и всем желающим их использовать, они должны удовлетворять определенным требованиям. Позволю себе сослаться в этом вопросе на несколько указаний, опубликованных в достаточно древней, но и до сих пор, на мой взгляд, полезной книге Лео Скэнлона [Л4-18].

- Документируйте макроопределения как можно тщательнее. Включайте побольше комментариев. Помните, смысл ваших макроопределений должен быть понятен любому, кто будет ими пользоваться, а не только вам.

- При вводе комментариев пользуйтесь макрооператором ;;.
- Старайтесь делать макроопределения как можно более универсальными. Если вам требуется специфичное макроопределение, то по возможности выражайте его через более универсальные макроопределения.
- Если макроопределение содержит метки, перечислите их в директиве **LOCAL**.
- Сохраняйте все используемые в макроопределении регистры, за исключением выходных. Как обычно, это делается с помощью команд **PUSH** в начале макроопределения и **POP** в конце.

 **ПРИМЕР 15.3-2**

Немного изменим пример (например, будем выдавать по три рожицы в строке) и подключим разработанные нами макросы, предварительно создав файл **MyMacro.inc**. Постараемся удовлетворить всем перечисленным выше требованиям: добавим комментарии, а некоторые макросы (**print_AL** и **move_cursor**) сделаем более общими (за счет употребления параметров). Исходный текст программы в результате уменьшится, а мы получим свою макробиблиотеку.

Исходный текст макробиблиотеки MyMacro.inc

```
comment @
___________________________________
        МАКРООПРЕДЕЛЕНИЯ
(c)Copyright (c) 1992, 2001 by Голубь Н.Г.
___________________________________
@
IdivW       MACRO     y, a, b
;; y=a/b
;; целочисленное деление для 16-разрядных знаковых данных
            push_regs <ax,bx,dx>
            Mov        ax,a
            Mov        bx,b
            CWD
            IDIV       bx
            Mov        y, ax
            pop_regs   <dx,bx,ax>
            EndM

MessageOut MACRO      StringName
;; Изобразить строку с сообщением DS:StringName
;; Признак окончания строки "$"
            Push       dx
            Lea        DX,DS:StringName
            MessageDX
            Pop        dx
            EndM

MessageDX   MACRO
;; аналог Procedure Output
;; Вывод строки символов
            Push     ax
            Mov      ah,9
            Int      21h
            Pop      ax
```

```
            EndM
push_regs   macro   reg_list
;; Сохранение значений регистров в стеке
  IFNB <reg_list>   ;;если список reg_list НЕ пуст
    irp reg,<reg_list>
      push reg
    endm
  ENDIF
            endm    ;;push_regs

pop_regs    macro   reg_list
;; Восстановление значений регистров из стека
  IFNB <reg_list>
    irp reg,<reg_list>
      pop reg
    endm
  ENDIF
            endm     ;;pop_regs

move_cursor      macro  Row, Col
;; Перемещение курсора:
;; номер строки ——> Row
;; номер столбца ——> Col
            push_regs <ax,dx>
            mov      DH,Row
            mov      DL,Col
            mov      ah,2 ;;функция установки позиции курсора
            int      10h
            pop_regs <dx,ax>
            endm

print_AL    macro Count, BackGround, Color
;; Изобразить символ, значение которого загружено в AL
;; Count — счетчик символов
;; BackGround — цвет фона
;; Color — цвет символа
            push_regs <ax,bx,cx>
            mov      cx,Count ;;изобразить Count символов
  ;;атрибут вывода: 16*фон+цвет символа
            mov      bl,16*BackGround+Color
  ;;функция вывода символа на экран в текущей позиции курсора
            mov      ah,9
            int      10h
            pop_regs <cx,bx,ax>
            endm
```

Исходный текст программы mMACRO.asm

```
                    title    По диагонали ТРИ рожицы
; по мотивам книги Л. Скэнлона, с. 254-255 & П. Абеля, гл.9
if1
  include MyMacro.inc
endif
;*************************************************************
;                          ПРОГРАММА
;*************************************************************
```

```
sstack    segment para stack "stack"
          db       64 dup ("stack")
sstack    ends
code     segment para "code"
diag_line proc   far
          assume  cs:code,ss:sstack
;организовать нормальный возврат в DOS
          push     ds
          mov      di,0
          push     di
;сохранить регистры
          push_regs <ax,bx,cx,dx>
          sti                   ;разрешить обработку прерываний
          mov      ah,15        ;получить номер активной страницы->BH
          int      10h
          mov      ah,0         ;задать режим изображения
          mov      al,3         ;цветной текстовый режим, 80*25
          int      10h
          mov      dx,0         ;строка и столбец 0
set:    move_cursor Dh,DL       ;переместить курсор
          mov      al,2         ;"рожица"
          print_AL  3,4,10      ; 4 — цвет фона, 10 — цвет символа
          inc      dh           ; по строке
          inc      dl           ; по столбцу
          inc      dl
          inc      dl
          cmp      dh,25
          jne      set
; Ждать нажатия любой клавиши
          Mov      ah,1
          Int      21h
; Восстановить регистры
          pop_regs <dx,cx,bx,ax>
          ret
diag_line endp
code     ends
          end      diag_line
```

Жирным шрифтом отмечены макрокоманды и изменения в макросах. На этапе компиляции (см. файл **Mmacro.lst**) вместо макрокоманд будут подключены *нужные* макросы и сделаны в коде соответствующие макрорасширения, хотя в подключаемой библиотеке есть и НЕ нужные для нашей задачи макросы.

15.5. Программа тестирования клавиатуры

Посмотрим теперь пример использования немного более сложных макросов: инициализация COM-формата, вызов из макроса процедуры, макрос завершения программы.



ПРИМЕР 15.4

Сделать программу тестирования клавиатуры.

Предлагаемая вашему вниманию программа разработана студентом второго курса ХАИ Артемом Астафьевым. Здесь совсем другой стиль программирования: не книжный, а свой, основанный на знаниях. Попробуйте в этой программе **самостоятельно** разобраться. Здесь

есть и известные вам прерывания, и порты. Программа написана в стиле TASM и достаточно лаконична. Обратите внимание на ОТСУТСТВИЕ ЗАГОЛОВКОВ ПОДПРОГРАММ, хотя сами подпрограммы имеются (я их для наглядности отчеркнула пунктиром) и к ним происходят нормальные обращения.

Автор настоящей книги только чуть-чуть разбавила оригинал программы комментариями, кое-что вставила и кое-что удалила (макробиблиотека оригинала была в 4 раза больше), сохранив по возможности стиль оригинала.

Исходный текст макробиблиотеки Ace

```
    %NOLIST
    ideal
;; ——— COM - формат (начало)
macro   begin
    model tiny
    locals      @@
    nojumps
segment code16     byte public     use16 'Code'
    assume      cs:code16,ds:code16,es:code16
    org  80h
label   cmdline    byte
cmdlen  db    ?
    org  100h
    p386
start:
endm
;;——— КОНЕЦ программы
macro   endprg
    ends
    end  start
endm
;;—— Установка видеорежима mode
macro   videomode mode
    mov        ax, mode
    int        10h
endm
;;— Вывод сообщения number
macro   message         number
    mov  bl,number
    call prnmsg
endm
    %LIST
```

Исходный текст программы GetKeys.asm

```
        include              ace
        p386
        begin
        videomode            3
        xor                  bl, bl
        mov                  ax, 1003h
        int                  10h
        messagelist=offset   msglst
        message              0
```

```
@@locloop:          mov         ah, 11h
                    int         16h
                    jz          @@nokey
                    mov         cx, 0100h
                    mov         dx, 0300h+79
                    mov         bh, 0d3h
                    call        cls
                    mov         ah, 10h
                    int         16h
                    cmp         ah, 01
                    jz          @@exit
                    cmp         al, 0e0h
                    jnz         @@notkeyp
                    message     4
@@notkeyp:          or          al, al
                    jnz         @@notext
                    message     5
@@notext:           message     1
                    push        ax
                    call        printhexb
                    message     3
                    pop         dx
                    call        putc
                    message     2
                    mov         al, ah
                    call        printhexb
@@nokey:            mov         ah, 2
                    int         16h
                    xchg        al, [byte        @@prevsw+1]
                    xor         al, [byte        @@prevsw+1]
                    jz          @@locloop
                    mov         bh, 9eh
                    mov         cx, 1800h
                    mov         dx, 1800h+79
                    call        cls
@@prevsw:           mov         al, 0
                    mov         bl, 0dh
@@checksw:          rol         al, 1
                    jnc         @@keyoff
                    call        prnmsg
@@keyoff:           dec         bl
                    cmp         bl, 5
                    jnz         @@checksw
                    jmp         @@locloop
@@exit:
                    mov         al, 0f3h
                    out         60h,al
                    mov         al, 0
                    out         60h, al
                    mov         si, offset       writebuf
                    mov         bl,12
@@writelp:          lodsb
                    mov         cl,al
                    xor         ch,ch
                    mov         ah,5
```

```
                int             16h
                dec             bl
                jnz             @@writelp
                ret
;----------------------------------------------------------------
prnmsg:         push            dx ax bx
                xor             bh, bh
                shl             bx, 2
                mov             ax, [messagelist+2+bx]
                inc             ax
                jz              @@nocur
                dec             ax
                mov             dx, ax
                call            gotoxy
@@nocur:        mov             dx, [messagelist+bx]
                mov             ah, 9
                int             21h
                pop             bx ax dx
                ret
;----------------------------------------------------------------
printhexb:      push            ax
                ror             al,4
                call            @@prhalf
                pop             ax
;----------------------------------------------------------------
@@prhalf:       and             al,0fh
                cmp             al,9
                jbe             @@1
                add             al,7
@@1:            add             al,'0'
                mov             dl,al
                call            putc
                ret
;----------------------------------------------------------------
putc:           push            ax
                mov             ah, 6
                int             21h
                pop             ax
                ret
;----------------------------------------------------------------
gotoxy:         push            bx
                mov             ah, 2
                xor             bh, bh
                int             10h
                pop             bx
                ret
;----------------------------------------------------------------
cls:            mov             ax, 0600h
                int               10h
                ret
;------------- Данные: ------------------------------------------
msglst  dw offset m0, 00c15h, offset m1, 0100h, offset m2, 013ah
        dw offset m3, 0120h, offset m4, 0218h, offset m5, 0315h
   dw offset m6, 1800h+00, offset m7, 1800h+10, offset m8, 1800h+20
   dw offset m9, 1800h+30, offset ma, 1800h+40, offset mb, 1800h+50
```

```
   dw offset mc, 1800h+60, offset md, 1800h+70
m0 db "Keyboard test by (c)Artyom Astafyev,1998. Exit - ,10,13,'$'
m1                db                "ASCII Code: $"
m2                db                "Scan Code: $"
m3                db                "Character: $"
m4                db                "Extended keystroke$"
m5                db        "The key was pressed on the keypad$"
m6                db                "RShift$"
m7                db                "LShift$"
m8                db                "Ctrl$"
m9                db                "Alt$"
ma                db                "Scroll$"
mb                db                "Num$"
mc                db                "Caps$"
md                db                "Insert$"
writebuf          db                "Coded by ACE"
                  endprg
```

Сделайте Com-файл (длина-то всего 514 байт!), запустите его на счет, и вы увидите, как компьютер воспринимает вводимые вами с клавиатуры символы.



Основы программирования Windows-приложений на Ассемблере

Люди часто начинают стремиться к великим целям, когда чувствуют, что им не по силам маленькие задачи. И не всегда безрезультатно...

Лев Шестов (1866-1938 гг.)

Идеи правят миром.

Платон (428 или 427 – 348 или 347 гг. до н.э.)

Операционные системы современной линейки Windows являются достаточно сложными операционными системами по сравнению с DOS. Они, как известно, очень требовательны к ресурсам компьютера, но очень нравятся пользователю.

Писать **подготовленному** программисту **прикладные программы (Applications — приложения)** на Ассемблере для Windows, как ни странно, проще, поскольку НЕ нужно изучать в деталях, как программировать **стандартные** устройства ввода-вывода компьютера на низком уровне, — для этого есть более 2000 системных функций (количество их постоянно растет). Эти функции достаточно подробно описаны в системах помощи (HELP — среда разработки Borland C++) и в постоянно пополняемой, колоссальной по объему библиотеке технической документации **MSDN-Library** (**MSDN** — Microsoft Developer Network) — основном источнике всех нововведений компании Microsoft.

Программирование же нестандартных устройств в Windows ***идейно*** *ничем НЕ отличается от DOS: те же прерывания, те же порты.* Но **Windows является многозадачной 32-разрядной операционной системой**. Поэтому, чтобы успешно программировать нестандартные устройства (обычно это делается на С и Ассемблере), нужно хорошо разбираться в особенностях 32-разрядного программирования (и особенно в защищенном режиме). **DDK** (Device Driver Kit) представляет собой богатый источник информации о внутренних структурах Windows и предназначен для разработчиков драйверов устройств. Он доступен для бесплатного скачивания на сайте Microsoft **www.microsoft.com/hwdev**.

Проще всего научиться Windows-программированию, используя ОЧЕНЬ дружелюбную и понятную среду программирования Borland (Inprise) Delphi и ее библиотеку визуальных компонентов (**VCL** — Visual Component Library). Далее, при желании, можно переключиться на Borland (Inprise) C++ Builder (последние версии поддерживают все стандартные Windows-библиотеки: VCL, MFC, OWL — Object Windows Library) и, наконец, вершина Windows-программирования для профессионалов — Microsoft Visual C++ и ее библиотека **MFC** (Microsoft Foundation Classes). Все эти библиотеки используют **объектно-ориентированный подход** к программированию (инкапсуляция, наследование, полиморфизм) и позволяют, используя базовые разработки в виде библиотек функций и ресурсов, менять интерфейс прикладной программы, как кому вздумается.

Освоившись с надводной частью этого айсберга под названием Windows, можно начать и ныряние в Ассемблер ... Оно ОЧЕНЬ интересное, поскольку Ассемблер позволяет избавиться в загрузочном модуле от массы избыточной информации, порождаемой компиляторами с языков высокого уровня, а также ПОЛНОСТЬЮ контролировать и оптимизировать программный код (кстати, в любой операционной системе!).

Функции **Win32 API** (Application Programming Interface — 32-разрядный интерфейс приложений) реализуют *нижний* Windows-уровень программирования. Эти функции используют **процедурный подход** к программированию. На них базируются функции других, объектно-ориентированных Windows-библиотек (MFC, VCL, OWL). Для каждой операционной системы линейки Windows реализуется **свое подмножество** Win32 API.

С описанием Win32 API и с примерами его использования на языке С можно познакомиться, установив SDK (Platform Software Development Kit). В его составе есть файл описания функций, расположенный в **\Microsoft Platform SDK\ Bin\Win32API.Txt,** размером более 700 Кбайт. SDK доступен для бесплатного скачивания на сайте Microsoft **http://msdn.microsoft.com/developer/sdk/.**

Функции **Win32 API** тоже постоянно обновляются и расширяются (как и сама Windows!), но компания Microsoft старается поддерживать совместимость своих функций сверху вниз.

Мы в данной главе познакомимся с основными принципами и идеями программирования Windows-приложений с помощью базовых функций **Win32 API. Практическую реализацию** этих идей — см. в лабораторной работе № 10.

16.1. Современные Windows-платформы

На сегодняшний день компания Microsoft поставляет операционные системы с **тремя ядрами (Windows 2000, Windows 98, Windows SE)**. Каждое ядро оптимизировано под свой класс задач.

Windows 2000 — это операционная система **класса "High-End"** и предназначена для **профессиональной работы** на рабочих станциях и серверах. Она обладает высокой степенью защищенности, полностью поддерживает Unicode, имеет богатый набор средств и утилит для администрирования системы, поддерживает многопоточность и мультипроцессорную обработку и не позволяет никому напрямую обращаться к оборудованию.

Windows 98 — операционная система **потребительского класса**, она менее защищена, поддерживает Unicode лишь частично, может работать только с одним процессором, но зато более дружественна к пользователю и разрешает ему беспрепятственно играть в компьютерные игры (что очень часто ведет к краху или зависанию операционной системы).

Windows 95 и **Windows Millennium Edition** имеют точно такое же ядро, как и **Windows 98. Windows XP** представляет собой компромисс между Windows 2000 и Windows 98, который позволяет сделать Windows 2000 более дружественной к пользователю.

Windows SE — это самое новое ядро, намного менее требовательное к памяти, поскольку эта операционная система рассчитана на карманные и автомобильные компьютеры, "интеллектуальные терминалы", торговые автоматы, микроволновые печи и проч.

Ядра этих операционных систем сильно отличаются друг от друга, поэтому **одни и те же функции Win32 API** соответствующих операционных систем **реализованы по-разному**, что и должен учитывать программист. Специфику их реализации на конкретной платформе можно посмотреть на сайте компании Microsoft **msdn.microsoft.com** или компакт-дисках **MSDN-Library**. Та же информация содержится в файле **Microsoft Platform SDK\Lib\Win32API.Csv**, который копируется при установке Microsoft Platform SDK. Этот файл входит и в поставку Microsoft Visual C++.

Основные системные библиотеки обычно находятся в папке **Windows\System (Winnt\System32, Win2k\System32)** и содержат большое количество библиотек Win32 API. В число **ключевых** входят три библиотеки:

- Библиотека ядра Windows (**kernel32.dll**).
- Библиотека пользователя (**user32.dll**).
- Библиотека графического интерфейса с устройствами (**GDI** — Graphical Device Interface — **gdi32.dll**).

Для получения полного списка доступных функций и услуг, предоставляемых любой из библиотек установленной у вас Windows, достаточно войти в нужную системную директорию и из командной строки вызвать программу фирмы Borland **impdef.exe**, чтобы создать текстовый **def**-файл (файл определения интерфейса). Например,

impdef.exe -a user32.def user32.dll

Среды разработки на языке C++, ориентированные на создание Windows-приложений (Microsoft Visual C++, Borland C++ Builder), содержат всю необходимую информацию о функциях **Win32 API** в своих системных заголовочных файлах. Основными файлами являются следующие: **windef.h, winbase.h, wingdi.h, winnt.h, winuser.h**. При программировании на Ассемблере достаточно трансформировать информацию, содержащуюся в этих файлах (а иногда и из других h-файлов), в ассемблеровский макрос.

 **ЗАМЕЧАНИЕ**

Windows 2000 на современном этапе представляет собой надмножество всех Win32-реализаций.

16.2. Типы данных

Компания Microsoft объявила официально о начале разработки графической операционной оболочки Windows 10 ноября 1983 года. Посмотрите, уважаемый читатель, в табл. 3.1, какими процессорами из семейства IBM PC располагали тогда программисты... А тут предлагаются грандиозные (и многим в то время непонятные!) идеи о многозадачности, графическом интерфейсе, независимости от типов дисплеев и принтеров! Поэтому неудивительно, что эта разработка вызвала неоднозначную реакцию и ожесточенные споры, как среди

пользователей, так и среди программистов. Многие даже считали, что компания Microsoft приняла недальновидное и ошибочное решение. Положение стало меняться с появлением в апреле 1992 году первой серьезной версии — Microsoft Windows 3.1 (пока все еще в ранге Win16-оболочки, реализованной в MS DOS). Тогда же начали появляться и более продвинутые процессоры, появился вкус и интерес к графике, а затем и к мультимедиа.

Реализована эта оболочка (и все предыдущие) была на языке Паскаль, поэтому в языковой среде Windows используются многие неявные соглашения для типов данных, типичные для Паскаля. Даже вызовы Windows-функций, составлявших основу Win16 API, при последующем переходе на язык C/C++ используют соглашение о связях языка Паскаль (см. п. 9.1.2.3) и записываются с помощью ключевого слова **PASCAL**. Преемственность Win16 API-Win32 API позволила максимально облегчить программистам-разработчикам бремя переноса существующих 16-разрядных Windows-приложений на новую 32-разрядную платформу и породила новый тип вызова функций в Ассемблере **STDCALL** (см. п. 16.6).

Типов данных в Win32 API достаточно много. С полным их перечнем можно познакомиться в справочной системе компании Microsoft — **MSDN-Library**. Все эти типы данных пишутся **ПРОПИСНЫМИ** буквами и присутствуют в макросах Windows (например, **Win32.inc**). Чтобы дать общее о них представление, перечислим некоторые из них.

Таблица 16.1. Основные типы данных Win32 API.

Тип данных	Эквивалент в C/C++	Описание
BOOL	**bool (ISO/IEC 14882 "Standard for the C++ Programming Language" - 1998 г.)**	Логическая переменная, возвращающая значение TRUE или FALSE
BOOLEAN		
BYTE		Байт - 8 бит
CHAR	**char**	8-битовый Windows (ANSI) символ.
CONST	**const**	
DWORD	**unsigned int**	32-разрядное беззнаковое целое
DWORD_PTR	**unsigned int***	Указатель (pointer) на 32-разрядное беззнаковое целое
DWORD32	**unsigned int**	32-разрядное беззнаковое целое
DWORD64		64-разрядное беззнаковое целое
FLOAT	**float**	
HANDLE	**unsigned int**	Дескриптор (номер) объекта: 32-разрядное беззнаковое целое.
HDC		Дескриптор контекста устройства (Handle to a device context - DC).
HWND		Дескриптор окна (Handle to a window).
INT	**int**	32-разрядное знаковое целое
INT_PTR	**int***	Указатель на 32-разрядное знаковое целое
INT32	**int**	32-разрядное знаковое целое
INT64		64-разрядное знаковое целое

LONG	int	32-разрядное знаковое целое
LONG_PTR	int*	Указатель на 32-разрядное знаковое целое
LONG32	int	32-разрядное знаковое целое
LONG64		64-разрядное знаковое целое
LONGLONG		64-разрядное знаковое целое
LPCSTR	const char*	Указатель на константную строку ANSI-символов в стиле C (завершающуюся символом '\0').
LPCTSTR	const w_char*	То же, что и **PCWSTR**
LPSTR	char*	Указатель на строку ANSI-символов в стиле C (завершающуюся символом'\0').
LPTSTR	w_char*	Указатель на Unicode-строку
PCWSTR	const w_char*	Указатель на константную Unicode-строку
PWSTR	w_char*	Указатель на Unicode-строку
POINTER_32	int*	32-разрядный указатель
POINTER_64		64-разрядный указатель
SHORT	short int	16-разрядное знаковое целое
UCHAR	unsigned char	
UINT	unsigned int	32-разрядное беззнаковое целое
UINT32	unsigned int	32-разрядное беззнаковое целое
UINT64		64-разрядное беззнаковое целое
ULONG	unsigned int	32-разрядное беззнаковое целое
ULONG32	unsigned int	32-разрядное беззнаковое целое
ULONG64		64-разрядное беззнаковое целое
ULONGLONG		64-разрядное беззнаковое целое
USHORT	unsigned short int	16-разрядное беззнаковое целое
VOID	void	
WCHAR	w_char	16-битный Unicode-символ
WORD	unsigned short int	16-разрядное беззнаковое целое

16.3. Соглашения об именах

Чтобы как-то систематизировать имена, используемые при разработке такого огромного продукта, как Windows, в компании Microsoft были приняты следующие соглашения для образования имен переменных, структур и функций:

- По возможности избегать сокращений в именах. Имена должны иметь смысловую нагрузку и отражать суть процесса или действия.
- Имена должны начинаться с большой буквы.
- Имена могут образовываться из двух (и более слов) определенным образом — см. табл. 16.2.

Таблица 16.2. Принцип образования имен в Windows.

Принцип	Пример
СуществительноеСуществительное	DialogBox, MessageBox
ГлаголСуществительное	LoadIcon, DrawText
ПрилагательноеСуществительное	FullName
Комбинация предыдущих грамматических элементов слов	MaximumApplicationAddress, ActiveProcessorMask, TotalPageFile, OpenCommPort, ReadIntervalTimeOut.

16.4. Венгерская нотация

Над кодом операционной системы Windows работало много программистов в течение многих лет. Код стремительно разрастался и программист фирмы Microsoft Чарльз Симонаи (Charls Simonyi), венгр по национальности, предложил систему префиксов (**всегда маленькая буква**) для обозначения имен переменных, связанную с их типом. В документации по Windows она широко используется, поэтому и мы познакомимся с ее основными идеями.

Таблица 16.3. Префиксы венгерской нотации.

Префикс	Тип данных	Пояснение
c	CHAR	
b	BOOL	
by	BYTE	
n	SHORT	
i	INT	
x	SHORT	Координата x
y	SHORT	Координата y
cx	SHORT	Относительная координата x
cy	SHORT	Относительная координата y
w	WORD	
l	LONG	
dw	DWORD	
u	unsigned	
h	HANDLE	Дескриптор (номер) объекта
fn		Указатель на функцию
sz		C-строка - строка с завершающим нулем (String ZERO)
st		Паскаль-строка
p		Указатель (Pointer)
lp		Указатель FAR (LONG Pointer)
np		Указатель ближний (Near Pointer)

Например, на языке C/C++ описание функции, используемой для вывода текста в окно сообщения (MessageBox), имеет следующий вид (см. **MSDN-Library**):

```
int MessageBox (
HWND hWnd, // handle to owner window — дескриптор окна владельца (0-владельца НЕТ)
LPCTSTR lpText, // text in message box — выводимый в окно текст
LPCTSTR lpCaption, // message box title — заголовок окна
UINT uType      // message box style — стиль окна
);
```

Все параметры являются входными **[in]**.

Зная префиксы, теперь мы можем читать (или хотя бы пытаться понять) MSDN, а значит, и Windows.

16.5. Получение EXE-файла

Исполняемые современные Windows-приложения должны разрабатываться как 32-разрядные, поэтому для их получения нужно использовать 32-разрядные версии компилятора (например, **tasm32.exe**) и соответствующую ему версию компоновщика (**tlink32.exe**). Все следующие ниже примеры были разработаны и отлажены в операционной системе **Windows 2000 Professional (SP2)** с использованием **Turbo Assembler Version 5.3** и **Turbo Link Version 1.6.71.0**, которые входят в поставку *Borland C++ Builder 5.0 Enterprise Suite.*

Windows-приложения должны работать в защищенном режиме. Поэтому они, в отличие от DOS-задач, имеют другой, **PE-формат** (Protection Enable — разрешение защищенного режима), которым обладают два вида файлов: **EXE-** и **DLL**-файл (Dynamically Linked Library — динамически подключаемая библиотека). EXE-файл может реализовывать два вида приложений: оконное или консольное. Все эти и другие нюансы нужно сообщить компоновщику.

Таблица 16.4. Значение ключей компоновщика (редактора связей) tlink32.exe.

Ключ	Значение
/Tpe	EXE-формат исполняемого файла
/Tpd	Формат исполняемого файла - DLL-библиотека
/aa	Оконное Windows-приложение
/ap	Консольное Windows-приложение
/c	Различать строчные и прописные буквы
/o	Обеспечить импорт библиотек

О том, что нужно различать строчные и прописные буквы, нужно сообщить и компилятору. Мы это уже делали при организации стыковки модуля на Ассемблере с модулем на языке C/C++ с помощью ключа **/ml**.

Теперь мы готовы к тому, чтобы создать пакетные командные файлы для получения EXE-файла: для оконного WIN32-приложения — **EXE32win.bat** и для консольного — **EXE32winConsole.bat**. Они будут **немного отличаться** от уже известного нам файла buildEXE.bat.

Текст пакетного командного файла EXE32win.bat (БЕЗ ресурсов)

```
@echo off
if -%1 == - goto NoParam
if -%2 == - goto NoParam
if not exist %1 goto NotExist
goto Ok
:NotExist
echo ERROR!!!
type %1
echo Not Exist!!!
goto exit
:NoParam
echo ERROR!!!
echo Variant of a call:
echo EXE32.bat Rewri.asm Rewri.exe
goto exit
:Ok
tasm32 %1 temp.obj /la /ml
if exist %2 del %2
tlink32 /Tpe /aa /c /o temp.obj
ren Temp.exe %2
if exist *.obj del *.obj >nul
if exist *.map del *.map >nul
:exit
```

Файл **EXE32winConsole.bat** будет отличаться от приведенного выше файла только одной строкой:

```
tlink32 /Tpe /ap /c /o temp.obj
```

Если приложение использует ресурсы (см. п. 16.6.1), то должен быть заранее подготовлен текстовый файл ресурсов (**rc-файл**), который нужно оттранслировать в двоичный файл (**RES-файл**) с помощью программы **brcc32.exe** и присоединить на этапе компоновки к obj-файлам.

Текст пакетного командного файла EXE32winRes.bat (С РЕСУРСАМИ)

```
@echo off
if -%1 == - goto NoParam
if -%2 == - goto NoParam
if -%3 == - goto NoParam
if not exist %1 goto NotExist1
if not exist %3 goto NotExist3
goto Ok
:NotExist1
echo ERROR!!!
type %1
echo Not Exist!!!
goto exit
:NotExist3
echo ERROR!!!
```

```
echo Not Exist!!!
goto exit

:NoParam
echo ERROR!!!
echo Variant of a call:
echo EXE32.bat Rewri.asm Rewri.exe Rewri.rc
goto exit

:Ok
tasm32 %1 temp.obj /la /ml
if exist %2 del %2
copy %3 temp.rc
brcc32 temp.rc
tlink32 /Tpe /aa /c /o temp.obj,,,,,temp.RES
ren Temp.exe %2
if exist *.obj del *.obj >nul
if exist *.map del *.map >nul
if exist temp.* del temp.* >nul
:exit
```

16.6. Оконное приложение

Оконное приложение в полной мере реализует все прелести графического режима Windows и базируется на использовании специального набора функций, составляющих графический интерфейс пользователя **GUI** (Graphic User Interface).

16.6.1. Основные определения

Процесс (Process) — приложение, находящееся в стадии выполнения. Оно имеет собственное виртуальное пространство, код, данные и другие ресурсы Windows. Каждый процесс должен иметь хотя бы один поток.

Поток (Thread) — отдельно выполняемая и управляемая часть процесса. Наличие потоков позволяет упростить решение многих сложных задач и усилить надежность работы приложений. В Visual C++, например, есть утилита **Spy++**, которая позволяет отслеживать все процессы и связанные с ними потоки.

Сообщение (Message) — способ передачи информации приложению. В Windows насчитывается более 900 сообщений, связанных с большим количеством событий (работа с клавиатурой, мышью, элементами управления окна и проч.). Все сообщения представляют собой константы и содержатся в файле библиотеки макросов Windows **Win32.inc** (или **winuser.h**). Чтобы их как-то различать и систематизировать, им дается префикс в виде двух заглавных латинских букв и подчерка. Перечислим некоторые из них:

- **WM_XXX** (Window State Message) — оконное сообщение. Например,
  WM_PAINT = 000Fh ;; перерисовать окно
  WM_QUIT = 0012h ;; выход
  WM_MOUSEACTIVATE = 0021h ;; активизировать мышку
- **EM_XXX** (Edit Message) — сообщения элементов управления окна редактирования. Например,
  EM_SCROLL = 00B5h ;; режим прокрутки
- **WS_XXX** (Windows styles) — стили окон. Например,

WS_POPUP = 080000000h ;; всплывающее окно
WS_VSCROLL = 000200000h ;; окно, с вертикальной полосой прокрутки
WS_HSCROLL = 000100000h ;; окно с горизонтальной полосой прокрутки

- **BM_XXX** (Button Control Messages) — сообщения элементов управления Button (кнопка). Например,

  #define BM_GETSTATE 0x00F2 // получить состояние
  #define BM_SETSTATE 0x00F3 // установить состояние

Ресурсы приложения (Resources) — массивы данных, которые программист может добавить в исполняемый модуль. При программировании, например, с использованием Visual C++ эти ресурсы могут быть описаны автоматически при проектировании приложения и сохранены в *текстовом файле* с расширением **rc** (rc-файл). Современные среды программирования под Windows имеют редакторы ресурсов, с помощью которых и готовится файл ресурсов. Его можно подготовить и вручную в обычном текстовом редакторе, а также изменить. Идентификаторы (Identifiers) стандартных ресурсов имеют префикс **ID_XXX**. Ресурсы бывают стандартные (rc-файл) и ресурсы программиста (rc2-файл).

Основные стандартные ресурсы:

- **Акселераторы** (Accelerators) — списки горячих клавиш (Hot-keys) и команд, ассоциированных с ними. Например, ресурс **ID_EDIT_COPY** связан с командой **Ctrl+C**.
- **Шаблоны панели инструментов** (Toolbars) — bmp-файлы для отображения на экране элементов управления. Например, описание таких ресурсов имеет следующий вид:

  BUTTON ID_EDIT_CUT
  BUTTON ID_EDIT_COPY
  BUTTON ID_EDIT_PASTE
- **Курсоры** (Cursors) — битовые массивы для отображения курсоров мышки.
- **Шаблоны диалогов** (Dialogs) — описание окна диалога (размер, стиль окна, расположение и стиль элементов управления, надписи и проч.).
- **Пиктограммы** (Icons) — битовые массивы (ico-файлы) для сокращенного представления некоторых элементов приложения (иконки).
- **Шаблоны меню** (Menu) — описание элементов меню. Например,
  MENUITEM "E&xit", ID_APP_EXIT

SDI-приложение (Single Document Interface) — приложение содержит основное окно, может иметь меню и работает с одним открытым документом (дочерним окном).

MDI-приложение (Multiple Document Interface) — приложение, в отличие от SDI-приложения, может работать с несколькими открытыми документами (дочерними окнами) — многооконное приложение.

Диалоговое приложение (Based Dialog) — приложение содержит только диалоговое окно с элементами управления. Оно не имеет основного окна, а значит, НЕ имеет меню.

16.6.2. Основы организации пользовательского интерфейса

Пользовательский интерфейс Windows разрабатывался, исходя из стандарта **SAA** (System Application Architecture — системная архитектура приложения), принятого фирмой IBM еще в ноябре 1988 г. Составная его часть **CUA** (Common User Access — единый доступ пользователя) описывала типы окон, их составные части и способы взаимодействия.

Окно (Window) — прямоугольная область экрана для организации обмена информацией между пользователем и приложением. Одновременно может быть доступно (активно) только одно окно и приложение, к которому это окно относится. Каждое окно имеет свой стиль, который характеризует тип и вид окна. Это константы с префиксом **WS_XXX** (Windows styles). Окно может быть **основное** (например, перекрывающееся — стиль **WS_OVERLAPPEDWINDOW**) и/или **вспомогательное** (всплывающее — стиль **WS_POPUPWINDOW**). *У приложения может быть только одно основное (главное) окно.*

Основное окно может иметь **дочерние** окна (стиль **WS_CHILDWINDOW**).

Вот как выглядит описание этих оконных стилей в системном заголовочном файле **winuser.h**

системный заголовочный файл winuser.h:

```
/*
 * Common Window Styles
 */
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED   | \
                 WS_CAPTION    | \
                 WS_SYSMENU    | \
                 WS_THICKFRAME  | \
                 WS_MINIMIZEBOX | \
                 WS_MAXIMIZEBOX)

#define WS_POPUPWINDOW   (WS_POPUP     | \
                 WS_BORDER     | \
                 WS_SYSMENU)

#define WS_CHILDWINDOW   (WS_CHILD)
```

А вот как описаны эти же стили в стандартном файле библиотеки макросов **Win32.inc**, входящем в поставку **Tasm5**:

```
; Common Window Styles
WS_OVERLAPPEDWINDOW = WS_OVERLAPPED OR WS_CAPTION OR
    WS_SYSMENU OR WS_THICKFRAME OR WS_MINIMIZEBOX OR
    WS_MAXIMIZEBOX
WS_POPUPWINDOW = WS_POPUP OR WS_BORDER OR WS_SYSMENU
WS_CHILDWINDOW = WS_CHILD
```

Вспомогательное окно может быть вторичным или диалоговым (DialogBox).

Диалоговое окно может быть модальным (Modal) или немодальным (Modeless).

Модальные диалоговые окна для продолжения работы приложения требуют от пользователя ввода данных или ответа на вопрос. Например, окно, содержащие различные управляющие элементы (см. рис. 16.1), будет модальным, поскольку требует ввода данных.

Немодальные диалоги могут использоваться для (постоянного) отображения какого-то процесса или информации на экране. Этих окон может быть несколько. Диалог может быть прерван в любое время. Окно, изображенное на рис. 16.2, может быть как модальным, так и не модальным (поскольку может НЕ требовать ответа для продолжения работы приложения).

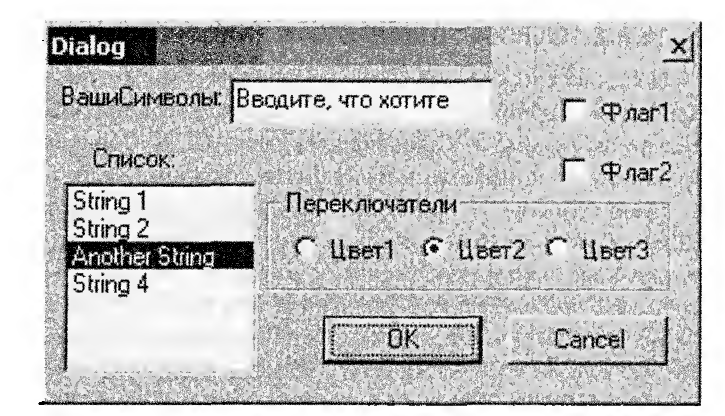


РИС. 16.1. *Пример модального диалогового окна с различными управляющими элементами.*

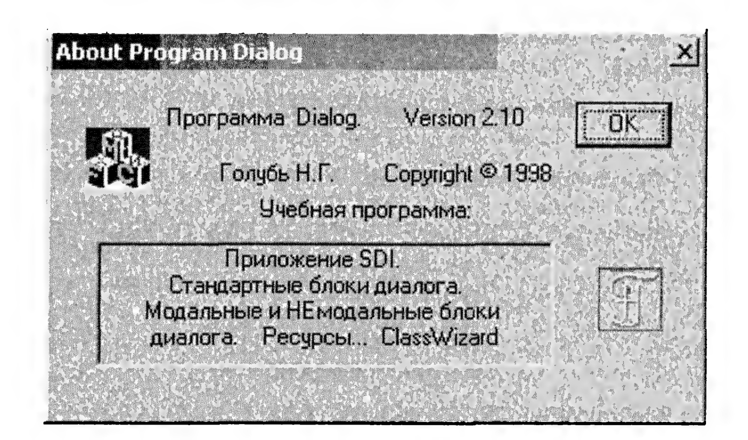


РИС. 16.2. *Пример диалогового окна с иконками и управляющими элементами.*

В Windows существует строгая иерархия окон. Каждое окно имеет родителя и ни одного или несколько элементов одного уровня.

Корнем является **окно рабочего стола**, создаваемого Windows при загрузке. Для Windows NT 4 и ее клонов может быть несколько рабочих столов одновременно.

Родительским окном для окон верхнего уровня является рабочий стол. Для дочернего окна родительским окном является окно верхнего уровня или другое дочернее окно более высокого уровня иерархии.

Для окон верхнего уровня с расширенным стилем **WS_EX_TOPMOST** нарушается видимая иерархия окон — они отображаются поверх всех остальных окон верхнего уровня без этого стиля.

Владелец (Owner) устанавливает связь между окнами одного уровня. Окно приложения является владельцем диалогового окна. Диалоговое окно всегда отображается поверх своего владельца и, как мы знаем, НЕ является дочерним окном.

Основное окно имеет рамку (Window Border) и различные области, которые частично учитываются стилями окна. Наличие этих областей является необязательным. Перечислим их сверху вниз и слева направо:

- Иконка приложения и системного меню (System Menu Icon), заголовок окна (Window Title Bar), обычные три кнопки из системного меню (Push Button).

- Строка меню (Menu).
- Панель инструментов (Tool Bar).
- Рабочая (клиентская) область окна (Client Area), где и происходят основные события приложения. У этой области справа может быть вертикальная полоса прокрутки, а снизу — горизонтальная (Scroll Bar). При Windows-программировании надо следить, чтобы при любых изменениях эта область перерисовывалась.
- Строка состояния (Status Line). Обычно здесь отражаются подсказки о том, что происходит с приложением, в каком режиме работы оно находится и прочее.

16.6.3. Минимальная Windows-программа

Чтобы немного понять основные принципы Windows-программирования, рассмотрим программу на языке С. Эта программа была взята автором из замечательной книги П. Нортона и Р. Макгрегора [Л7-6], немного изменена и дополнена. Анатомия этой программы, написанной в процедурном стиле (**стиль SDK**) с использованием функций Win32 API, по мнению автора наиболее близко соответствует стилю языка Ассемблера и отражает основополагающие идеи простейшего Windows-программирования:

- главное окно (**hwndMain**), его описание и инициализация;
- модальные диалоговые окна (создаются функцией **MessageBox**);
- сообщения от мышки (**WM_LBUTTONDOWN, WM_RBUTTONDOWN**) и от пользователя о закрытии главного окна приложения (**WM_DESTROY**);
- обработка этих сообщений в оконной процедуре (**WndProc**).

Попробуйте с этой программой разобраться, получить asm-файл:

bcc32 -S MINSDK2.C.

Все ресурсы в этой программе (иконка приложения, кнопки) стандартные, поэтому файл ресурсов отсутствует.

Главной функцией является функция **WinMain**. Ее назначение:

- Инициализировать приложение.
- Инициализировать и создать окна приложения.
- Ожидать сообщения из очереди сообщений для своего окна.

Windows работает, используя принцип событийно-зависимого выполнения программ: есть событие в данном окне, значит, оно должно быть обработано оконной процедурой или переслано в Windows для стандартной обработки. Нет события — "спим" ...

Исходный текст программы SDKWIN2.C

```
/*****************************************************************
 Module   : SDKWIN2.C
  Реализация минимального Win32-приложения с использованием Win32
  API.
  Author   : Rob McGregor, rob_mcgregor@compuserve.com
  Date     : 02-23-96
  Реализация в среде Borland C++ 5.02
*****************************************************************/
```

```
#include <windows.h>
//
// Function prototypes — прототипы функций
//
INT PASCAL WinMain(HINSTANCE, HINSTANCE, LPSTR, INT);
BOOL InitApplication(HINSTANCE);
BOOL InitInstance(HINSTANCE, INT);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//
// Global variables — глобальные переменные
//
HINSTANCE ghInst; // current instance — текущий экземпляр приложения
char szAppName[] = "WinSdk2";    // The app name — имя приложения
// caption text — текст заголовка окна
char szAppTitle[] = "Minimal SDK Application";
/*********************************************************************
 Function : WinMain(HINSTANCE, HINSTANCE, LPSTR, INT)
 Purpose : Program entry point. Calls initialization function,
       processes message loop.
 Назначение: точка входа программы. Вызывает функцию инициализации,
       обрабатывает цикл сообщений.
*********************************************************************/

INT PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
          LPSTR lpszCmdParam, INT nCmdShow)
{
  MSG message;  // window message — оконное сообщение

  //
  // If no previous instance register the new window class
  // Если нет экземпляров приложения, зарегистрировать новый оконный класс
  //
  if (!hPrevInstance) // Are other instances of the app running?
  // Выполняются ли другие экземпляры приложения?
  {
   // Initialize 1st instance — инициализация первого экземпляра
   if (!InitApplication(hInstance))
   {
  // Exits if unable to initialize — выход при невозможности инициализации
     return (FALSE);
   }
  }

  // Perform initializations that apply to a specific instance
  // Выполнить инициализацию приложения со стилем окна
  // SW_SHOWMAXIMIZED
  if (!InitInstance(hInstance, SW_SHOWMAXIMIZED))
  {
   return (FALSE);
  }

  //
  // Enter the application message loop. Get and dispatch
  // messages until a WM_QUIT message is received
  // Войти в цикл сообщений приложения. Получать и посылать
  // сообщения до тех пор, пока не будет получено
```

```
   // сообщение выхода из приложения WM_QUIT
   //
   while (GetMessage(&message, NULL, 0, 0))
   {
    // Translates virtual key codes
    // Транслировать виртуальные коды ключей
    TranslateMessage(&message);
    // Dispatches message to window — направить сообщение окну
    DispatchMessage(&message);
   }
// Returns value from PostQuitMessage — возвратить значение
//                      из сообщения PostQuitMessage
   return (message.wParam);
}
/*****************************************************************
 Function : InitApplication(HINSTANCE)
 Purpose : Initializes window data and registers window class.
 Назначение: Инициализация данных окна и регистрация оконного
   класса
*****************************************************************/

BOOL InitApplication(HINSTANCE hInstance)
{
   WNDCLASS wc;   // Структура для описания любого окна

   //
   // Fill in window class structure with parameters
   // that describe the main window
   // Заполнение структуры оконного класса параметрами,
   // описывающими главное окно
   //
   wc.style          = CS_HREDRAW | CS_VREDRAW;// стиль окна
   wc.lpfnWndProc    = (WNDPROC)WndProc;     // оконная процедура
   wc.cbClsExtra     = 0;      // дополнительные байты класса
   wc.cbWndExtra     = 0;      // дополнительные байты окна
   wc.hInstance      = hInstance;   // дескриптор экземпляра окна
                                    // дескриптор пиктограммы
   wc.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
   wc.hCursor        = LoadCursor(NULL, IDC_ARROW);   // дескриптор
                                                      // курсора
   wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1); // цвет фона
   wc.lpszMenuName  = NULL;        // ресурс меню, связанный с окном
   wc.lpszClassName = szAppName;   // имя класса окна
   // Register the window class and return success or failure
   // Регистрация класса окна и возвращение результата регистрации:
   //              (успех или неудача)
   return (RegisterClass(&wc));
}

/*****************************************************************
 Function : InitInstance(HINSTANCE, int)
 Purpose : Saves instance handle and creates main window
 Назначение: Сохраняет дескриптор экземпляра и создает главное окно
*****************************************************************/

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
```

```
{
  HWND hwndMain;    // Main window handle — дескриптор главного окна.

  // Store instance handle in global variable
  // Сохранение дескриптора экземпляра в глобальной переменной
  ghInst = hInstance;

  //
  // Create a main window for this application instance.
  // Создание главного окна для этого экземпляра приложения.
  //
  hwndMain = CreateWindow(
   szAppName,           // window class name — имя оконного класса
   szAppTitle,          // window caption — заголовок окна
   WS_OVERLAPPEDWINDOW, // window style — стиль окна
// initial x position — инициализация начальной позиции окна по
                        //горизонтали x
   CW_USEDEFAULT,
// initial y position — инициализация начальной позиции окна по
                        // вертикали y
   CW_USEDEFAULT,
   CW_USEDEFAULT, // initial x size — инициализация размера по x
   CW_USEDEFAULT, // initial y size — инициализация размера по y
   NULL,          // parent window handle — дескриптор родительского окна
   NULL,          // window menu handle — дескриптор оконного меню
// program instance handle — дескриптор экземпляра приложения
   hInstance,
   NULL           // creation parameters — параметры создания окна
  );

  //
  // Return FALSE if window handle is NULL
  // (window not could not be created)
  // Если невозможно создать окно, вернуть FALSE
  //
  if (!hwndMain)
  {
   return (FALSE);
  }

  //
  // Make the window visible and update its client area
  // Сделать окно видимым и обновить его клиентскую область
  //
  ShowWindow(hwndMain, SW_SHOWMAXIMIZED);    // Show the window -
                                             //показать окно
  // Sends WM_PAINT message — послать сообщение WM_PAINT
  UpdateWindow(hwndMain);
  return (TRUE);                             // Success! — Успех!
}

/*****************************************************************
 Function : WndProc(HWND, UINT, WPARAM, LPARAM)
 Purpose : Processes messages
 Назначение: Обработка сообщений
*****************************************************************/

LRESULT CALLBACK WndProc(HWND hwndMain,      // window handle -
                                             //дескриптор окна
```

```
                UINT message,  // type of message - тип сообщения
                     // additional information - доп.информация
                WPARAM wParam,
                LPARAM lParam) // additional information.
{
  //
  // Some local variables
  // Локальные переменные
  //
  char szAboutLeft[] = //"This is a minimal Windows SDK program.\n"
                       //"You've pressed the left mouse button!";
           "Это - минимальное Windows SDK-приложение.\n"
           "Вы нажали левую кнопку мыши!";
  char szAboutRight[] = //"This is a minimal Windows SDK program.\n"
                        //"You've pressed the right mouse button!";
           "Это - минимальное Windows SDK-приложение.\n"
           "Вы нажали правую кнопку мыши!";
  //
  // message handlers
  // обработчики сообщений
  //
  switch (message)
  {
   // left mouse button pressed - нажата ЛЕВАЯ кнопка мыши
   case WM_LBUTTONDOWN:
   {
     MessageBeep(MB_ICONINFORMATION); // Подключение динамика
     MessageBox(GetFocus(), szAboutLeft,
           "Окно About - ЛЕВАЯ кнопка мыши",
           MB_OK | MB_ICONINFORMATION);
     break;
   }

   // right mouse button pressed - нажата ПРАВАЯ кнопка мыши
   case WM_RBUTTONDOWN:
   {
     MessageBeep(MB_ICONINFORMATION);
     MessageBox(GetFocus(), szAboutRight,
           "Окно About - ПРАВАЯ кнопка мыши",
           MB_OK | MB_ICONQUESTION);
     break;
   }

   // the window is being destroyed - разрушение окна
   case WM_DESTROY:
   {
     PostQuitMessage(0);
     return 0;
   }
   default:    // какие-то другие действия пользователя
     break;
  }

  //
  // Send unhandled messages off to Windows for processing
  // Переслать необработанные сообщения обратно Windows
  //
  return (DefWindowProc (hwndMain, message, wParam, lParam));
}
```

Чтобы эта программа заработала, воспользуемся средой программирования Borland C++ 5.02. Сделаем проект с динамическим подключением библиотек (**dll**) — см. рис. 16.3.

Запустим программу на выполнение и получим простейшее оконное приложение Minsdk2.exe с главным окном, содержащим только иконку приложения и системного меню (System Menu Icon), заголовок окна (Window Title Bar), обычные три кнопки из системного меню (Push Button) — см. рис. 16.4.

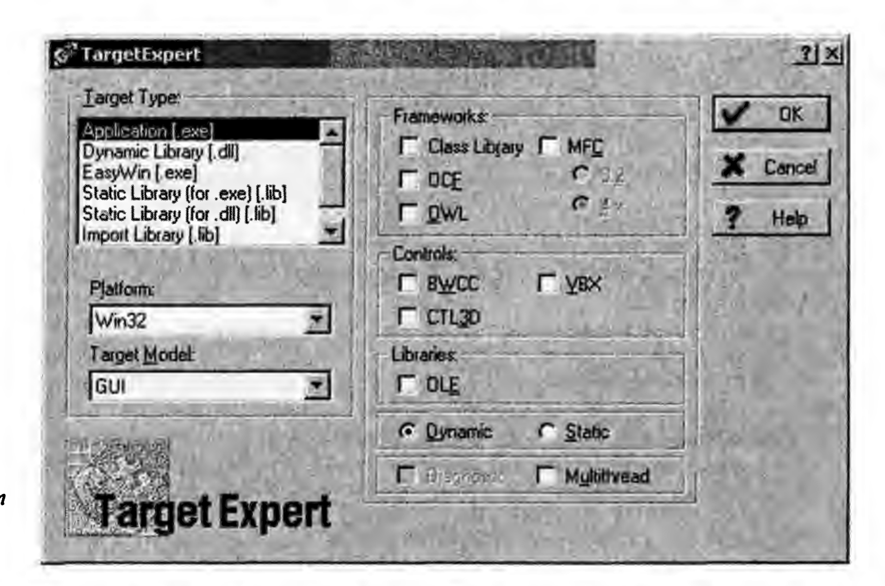
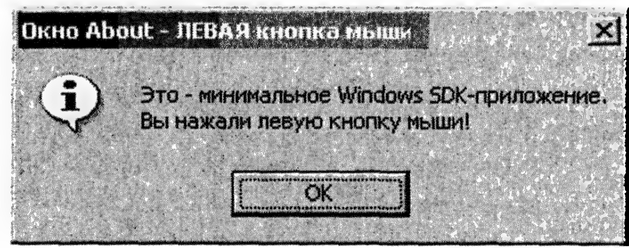


РИС. 16.3. *Проект для программы Minsdk2.c.*

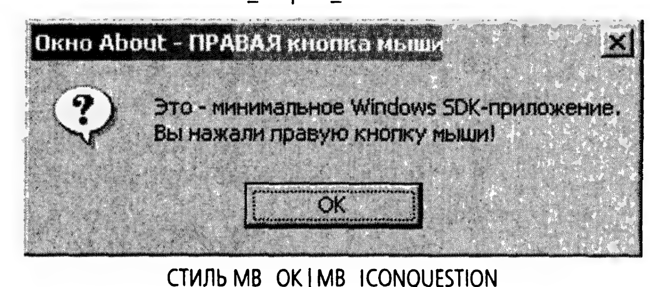
Если щелкнуть правой или левой кнопкой мышки, то получим немного разные диалоговые окна в соответствии с заданным стилем окошка — см. рис. 16.5. Кроме того, если к вашей Windows подключены **разнообразные** звуки, по-разному откликающиеся на разные события, то и они при выполнении приложения будут разные, хотя мы в программе указали один и тот же стиль звука MessageBeep(MB_ICONINFORMATION).



РИС. 16.4. *Главное окно приложения Minsdk2.exe.*



СТИЛЬ MB_OK | MB_ICONINFORMATION



СТИЛЬ MB_OK | MB_ICONQUESTION

РИС. 16.5. *Диалоговые окна, полученные с помощью функции MessageBox.*



ОБРАТИТЕ ВНИМАНИЕ

на то, что эти диалоговые окна модальные. Мы ничего НЕ сможем сделать с приложением Minsdk2.exe, пока не закроем эти окошки. Кстати, стиль кнопки MB_OK можно было и не указывать — он все равно будет создан по умолчанию. А вот, какие поясняющие значки и надписи в этом окошке, — это в ваших руках!

Теперь, уважаемый читатель, вы можете сами сделать asm-файл (или посмотреть его в прилагаемых к книге материалах) и начать анализ. Практически все команды вам знакомы.

16.6.4. Базовая структура модуля на Ассемблере

Структура модуля на Ассемблере при программировании Windows-приложения в формате EXE-файла может иметь следующий вид (см. рис. 16.6).

```
.386 ; .386p или старше — см. п. 12.1
.MODEL Flat, STDCALL
include     win32.inc   ; Windows-макробиблиотека
       ......
includelib  import32.lib ; Windows-библиотека
.DATA
       ; Данные
       ......
.CONST
       ; Константы
       ......
.CODE
ТочкаВхода:
       ; код
       .....
        end ТочкаВхода
```

РИС. 16.6. *Базовая структура модуля на Ассемблере при программировании оконных Windows-приложений.*

В этой структуре нам все понятно и привычно, кроме выделенных трех строк, а точнее одной (**.MODEL Flat, STDCALL**), т.к. о подключении библиотек макросов и библиотечных файлов мы уже упоминали в п. 15.3.8.

.MODEL Flat, STDCALL — эта директива, как мы знаем, обеспечивает плоскую модель (FLAT), которая используется для платформы Win32. А вот что такое **STDCALL**? Здесь тоже все достаточно просто. Платформа Win32 использует исключительно тип вызова функций STDCALL, который является гибридом между Паскалем и С: параметры передаются по порядку справа налево, как в языке С/С++ (см. п. 9.1.2.4), а по окончании работы вызываемая процедура должна сама очистить стек, как в Паскале (см. п. 9.1.2.3).

Рассмотрим на примере некоторые нюансы Windows-программирования на Ассемблере.



ПРИМЕР 16.1

Создать простейшее оконное приложение, которое должно вывести текстовую строку с сообщением "Я учусь программировать в Windows!".

Для решения этой задачи достаточно сделать *диалоговое приложение*. Воспользуемся уже немного знакомой нам функцией **MessageBox** (см. п. 16.4) и функцией **ExitProcess**, которая заканчивает процесс и все его потоки — см. **MSDN-Library**:

```
VOID ExitProcess(
  UINT uExitCode // exit code for all threads — завершающий код для всех потоков
);
```

Исходный текст программы Messbox.asm — вариант 1

```
; CopyRight by Голубь Н.Г., 2001
.386
.model flat, stdcall
; Windows-макробиблиотека (с добавлениями – Патлань Игорь,2001)
include  win32.inc
includelib import32.lib

.data
mesbox_text        db 'Я учусь программировать в Windows!',0
mesbox_title       db 'Заголовок окошка MessageBox',0

.code
start:
  ; Загрузка в стек параметров (задом наперед), как в C/C++:
  push  0 ; стиль окна (UINT uType) – по умолчанию
  push  offset mesbox_title; адрес заголовка окна (LPCTSTRlpCaption)
  push  offset mesbox_text ; адрес выводимого текста (LPCTSTR lpText)
  push  0 ; HWND hWnd=0-владельца нашего окна НЕТ
  ; Вызов функции MessageBox:
  call  MessageBox
  push  0 ; завершающий код для всех потоков (UINT uExitCode)
  call  ExitProcess
ends
end start
```

Создадим теперь EXE-файл оконного приложения:

EXE32win.bat MESSBOX.ASM MESSBOX.exe

и запустим его на выполнение. Мы получим диалоговое окно, изображенное на рис. 16.6.



РИС. 16.6. *Результат работы программы Messbox.asm.*

Если мы воспользуемся макробиблиотекой **win32.inc** из поставки TASM-5 (его длина 13042 байт), то мы получим сообщение об ошибке:

```
**Error** MESSBOX.ASM(19) Undefined symbol: MessageBox
**Error** MESSBOX.ASM(21) Undefined symbol: ExitProcess
```

В данном макросе НЕТ описания функций, а есть только константы. Поэтому студент второго курса ХАИ Патлань Игорь, имея некоторый опыт программирования в Borland Delphi и Microsoft Visual C++, проанализировал справочную информацию в MSDN, поставляемые фирмой Borland примеры и заголовочные файлы Windows (см. п. 16.1). И при реализации лабораторной работы № 10 он сделал в макробиблиотеку **win32.inc** (ее длина стала 29896 байт) необходимые вставки. Для нашего примера из этого макроса интерес представляют следующие строки:

```
.....................................................
;*****************************************************
;
;     Window's externals
;
;*****************************************************
.....................................................
extrn   ExitProcess:PROC
extrn   MessageBoxA:PROC
.....................................................
MessageBox     equ <MessageBoxA>
```

По поводу замены имени функции MessageBox на имя ее псевдонима (Alias — псевдоним) MessageBoxA имеются указания в нескольких файлах из тех, что мы уже перечисляли выше (**Win32API.Txt, user32.def** и **winuser.h**). Например, в заголовочном файле **winuser.h** есть такая строка:

```
#define MessageBox MessageBoxA
```

Теперь, зная все это, можно наш простой пример решить вовсе без макробиблиотеки **Win32.inc**, поскольку никаких Windows-констант мы НЕ используем.

Исходный текст программы Messbox.asm — вариант 2

```
   ; CopyRight by Голубь Н.Г., 2001
   .386
   .model flat, stdcall
   includelib   import32.lib

   extrn        MessageBoxA:PROC
   extrn        ExitProcess:PROC

   .data
   mesbox_text       db 'Я учусь программировать в Windows!',0
   mesbox_title      db 'Заголовок окошка MessageBox',0
   .code
   start:
; Загрузка в стек параметров (задом наперед), как в C/C++:
      push 0 ; стиль окна (UINT uType) — по умолчанию
      push offset mesbox_title ; адрес заголовка окна (LPCTSTR lpCaption)
      push offset mesbox_text ; адрес выводимого текста (LPCTSTR lpText)
      push 0 ; HWND hWnd=0 — владельца нашего окна НЕТ
; Вызов функции MessageBox:
      call MessageBoxA
      push 0 ; завершающий код для всех потоков (UINT uExitCode)
      call ExitProcess
ends
end start
```

 **ЗАМЕЧАНИЕ**

Второй вариант даст еще и EXE-файл длиной 4К байт против 8К байт в первом варианте.

Более подробные сведения о реализации диалогового приложения и связанных с ним ресурсах см. в лабораторной работе № 10.

16.7. Консольное приложение

Почему-то укоренилось мнение, что консольное приложение может работать только в текстовом режиме. Ниже будет приведен пример, опровергающий это мнение. Консольное приложение ОЧЕНЬ гибкое. Оно позволяет использовать в окне наряду с текстовым вводом-выводом еще и графический. И здесь, как и в оконном приложении, есть два уровня программирования:

- верхний — с помощью функций уже упоминавшихся объектно-ориентированных библиотек;
- нижний — с помощью функций Win32 API.

ПРИМЕР 16.2

Показать с помощью функций Win32 API возможности консольного приложения: символьный ввод-вывод (строки, числа) и графический вывод (эллипс с изменением в цикле цвета пера и палитры).

Сделаем консольное приложение в Visual C++ 6. Попробуем в нем разобраться. Ключевыми здесь являются три выделенные жирным шрифтом функции. Одна из них — "хитрая" системная функция **FindWindow**, которая открывает для консольного приложения все необходимые ресурсы, в том числе и собственное 32-разрядное виртуальное адресное пространство, обеспечивающее доступ к 4 Гбайт памяти.

Исходный текст программы 000.срр

```
#include <windows.h>
#include <iostream.h>
// Текстовый и графический вывод в консольном приложении
// Использование SDK Win32
// Идея — Лучшев П.А., 2001
// Реализация Голубь Н.Г., 2001.
#pragma once

int main(int argc, char* argv[])
{
  // Создание и регистрация окна консольного приложения
  HWND hWnd = FindWindow("ConsoleWindowClass",NULL);
  // Получение дескриптора контекста устройства для графического вывода
  HDC hDC   = GetDC(hWnd);
  cout << "\n !!!!!! OutPut Ellipse (ConsoleWindowClass) !!!!!!\n";
  cout << "\n    Enter any Number, and see....\n\n";
  cout << "\n    Exit - Number 0 or Ctrl/C \n\n\n";
  cout.flush();   // освобождение тестового буфера!!!!!!

  int k=0;
  int nR,nG,nB;
  HPEN hPen;      // описание пера
  HBRUSH hBrush; // описание кисти для закрашивания эллипса
  do
  {
  // Создание пера толщиной 10 (сначала зеленого цвета).
  nR=k;
  nG=255-10*k;
  nB=10*k;
  hPen = CreatePen(PS_SOLID, 10, RGB(nR, nG, nB));
  // Связывание пера с объектом вывода
  SelectObject(hDC, hPen);

  // Создание сначала красной кисти
  nR=255-20*k;
  nG=10*k;
  nB=200-k;
  hBrush = CreateSolidBrush(RGB(nR, nG, nB));
  // Связывание кисти с объектом вывода
  SelectObject(hDC, hBrush);
  // Рисуем эллипс
  Ellipse(hDC,20,40,100,100);

  // ждем ввода числа
  cin >> k;
  // перерисовываем окно
  UpdateWindow(hWnd);
  } while (k != 0);
```

```
    // Очистка памяти
    DeleteObject(hBrush);
    DeleteObject(hPen);
    return 0;
}
```

Если запустить это приложение на выполнение, мы увидим следующую **цветную** картинку — см. рис. 16.7.

Сделайте asm-файл и посмотрите, как он реализован. В общем, ничего нового с точки зрения Ассемблера...



ЗАМЕЧАНИЕ.

Очень советую с этим приложением поэкспериментировать самостоятельно — могут быть неожиданные и интересные эффекты (например, при нажатии клавиши **F2** и других, не числовых клавиш). Особенно в **разных** операционных системах линейки Windows. Посмотрите, как работает функция обновления окна. Попробуйте все это осмыслить и объяснить.

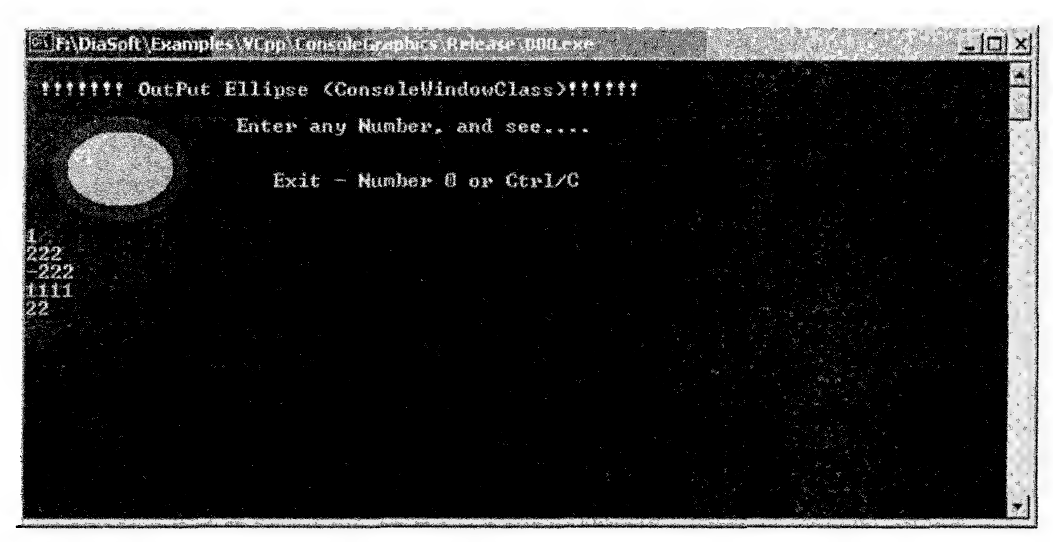
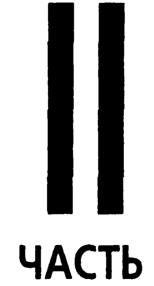


РИС.16.7. *Результат выполнения консольного Win32 приложения.*



ЧАСТЬ

Лабораторный практикум

Чтобы развиваться, нам надо учиться и заимствовать, никогда не страшась утратить нашу индивидуальность.

С.Н. Рерих (1904-1993 гг.)

ВВЕДЕНИЕ

Предлагаемый лабораторный практикум содержат перечень вариантов лабораторных работ по изучаемому курсу Ассемблера. Каждая работа имеет подробное решение типового варианта. Материал, данный в приложениях, ориентирован на язык ассемблера персональных ЭВМ на базе центрального процессора Intel x86. С целью более глубокого изучения принципов работы ЭВМ желательно знать форматы машинных команд, уметь *самостоятельно* делать ассемблирование и дизассемблирование, руководствуясь материалами приложений.

Предполагается, что все работы, кроме **первой, второй** и **десятой**, должны быть выполнены на ЭВМ с использованием любого **базового** алгоритмического языка (**PASCAL** или **C/C++**) и языка ассемблера (**TASM** или **MASM**). Заданный вариант задачи реализуется в виде подпрограмм-модулей на языке ассемблера и на выбранном алгоритмическом языке высокого уровня (для контроля правильности вводимых данных и анализа результатов). Затем из главной программы, написанной на алгоритмическом языке, осуществляется вызов сначала модуля на ассемблере и вывод результатов, а затем модуля на алгоритмическом языке. Результаты сравниваются и делаются соответствующие выводы.

Лабораторные работы **первая** и **вторая** выполняются вручную и на ЭВМ, с использованием компилятора с языка ассемблера (для контроля полученного результата). Лабораторная работа № 10 делается целиком на языке ассемблера.

В конце каждой лабораторной работы дается ориентировочный перечень вопросов для самоподготовки и самопроверки.

1

Лабораторная работа № 1

Внутреннее представление целочисленных данных в IBM PC

Труден лишь первый шаг.

Варрон Марк Теренций
(116-27 гг. до н.э.)

Цель работы

Выполнить перевод заданных преподавателем чисел из десятичной в двоичную систему счисления. Дать их внутреннее (машинное) представление в соответствии с диапазоном в знаковых и беззнаковых форматах типов **ShortInt (signed char)**, **Byte (unsigned char)**, **Integer (int)**, **Word (unsigned int)**. Машинное представление данных должно быть дано в двоичной и шестнадцатеричной системах счисления.

Порядок работы

1) вычислить для своего варианта целые числа;

2) перевести их из 10-тичной в 2-ичную (или 16-ричную) систему счисления;

3) получить их внутреннее представление;

4) написать программу описания этих чисел на языке Ассемблера и получить листинг;

5) проверить правильность своих выкладок.

Целочисленные данные должны быть представлены во всех возможных для платформы **Win16 (Win32)** форматах с учетом их диапазона представления (см. прил.4 — табл. П4.1-П4.4).

В отчете по лабораторной работе должен быть представлен подробный протокол перевода всех заданных чисел из 10-тичной в 2-ичную и 16-ричную системы счисления (**теоретическое введение**).

Варианты

Преподаватель задает два базовых числа ± **X** и ± **Y**. Студент должен прибавить и отнять от них № своего варианта.

Например, пусть **X =± 4567, Y =± 60, № = 45**. Тогда получатся следующие восемь целых чисел для варианта **№ = 45**, а именно:

1)	4567	+ 45	=	4612;
2)	4567	— 45	=	4522;
3)	— 4567	+ 45	=	— 4522;
4)	— 456	— 45	=	— 4612;
5)	60	+ 45	=	05;
6)	60	— 45	=	15;
7)	— 60	+ 45	=	— 15;
8)	— 60	— 45	=	— 105.

Контрольные вопросы

1. Перечислите все базовые целочисленные типы данных в IBM PC.
2. Диапазон допустимых значений для целочисленных данных.
3. Как получить внутреннее представление заданных данных в формате **LongInt (long int)**?
4. Что может означать, например, машинное представление 0F654h, и в каких форматах?

При выполнении этой лабораторной работы мы воспользуемся материалом, изложенным в пп. 1.2.1, 2.2 и в прил.4.

Пример решения типового варианта

Вариант №19. Базовые числа: X = ± 2235, Y = ± 40.

40 + 19 = 59	-40 + 19 = -21
40 − 19 = 21	-40 − 19 = -59
2235 + 19 = 2254	-2235 + 19 = -2216
2235 − 19 = 2216	-2235 − 19 = -2254

Решение

Для перевода целого числа, представленного в десятичной системе счисления, в систему счисления с основанием **q** необходимо данное число делить на основание до получения целого остатка, меньшего **q**. Полученное частное снова необходимо делить на основание до получения целого остатка, меньшего **q**, и т. д. до тех пор, пока последнее частное будет меньше **q**. Число в системе счисления с основанием **q** представится в виде упорядоченной последовательности остатков деления в порядке, обратном их получению. Причем старшую цифру числа дает последний остаток, а младшую — первый.

Для того чтобы сменить знак числа, нужно инвертировать все его биты и прибавить к нему единицу — получим представление ОТРИЦАТЕЛЬНОГО числа в дополнительном коде.

Для целых типов со знаком, под знак отводится старший бит, причем для положительных чисел он равен 0, а для отрицательных — 1.

Целочисленные типы, их диапазоны значений и количество требуемой для них памяти приведены в следующей таблице:

Тип	Диапазон значений	Требуемая память
Byte	0..255	1 byte
Shortint	-128..127	1 byte
Word	0..65535	2 bytes
Integer	-32768..32767	2 bytes
Longint	-2147483648..2147483647	4 bytes

Протокол перевода чисел

Используя приведенные в п.1.2.1 правила, сначала переведем заданные числа в двоичную систему счисления, а затем дадим их внутреннее представление. Результаты сведем в табл. 1:

59	2				
58	29	2			
1	28	14	2		
	1	14	7	2	
		0	6	3	2
			1	2	1
				1	

59d ==> **0011.1011b (BYTE)**

-59d ==> 1) 0011.1011b — двоичный код числа |-59|

2) 1100.0100b — инверсия

3) + 1

1100.0101b — **дополнительный код**

21	2			
20	10	2		
1	10	5	2	
	0	4	2	2
		1	2	1
			0	

21d ==> **0001.0101b (BYTE)**

-21d ==> 1) 0001.0101b — двоичный код числа |-21|

2) 1110.1010b — инверсия

3) + 1

1110.1011b — **дополнительный код**

2254	2										
2254	1127	2									
0	1126	563	2								
	1	562	281	2							
		1	280	140	2						
			1	140	70	2					
				0	70	35	2				
					0	34	17	2			
						1	16	8	2		
							1	8	4	2	
								0	4	2	2
									0	2	1
										0	

2254d ⟹ **0000.1000.1100.1110b (WORD)**

-2254d ⟹ 1) 0000.1000.1100.1110b — двоичный код числа |-2254|

2) 1111.0111.0011.0001b — инверсия

3) + 1

1111.0111.0011.0010b — ***дополнительный код***

2216	2										
2216	1108	2									
0	1108	554	2								
	0	554	277	2							
		0	276	138	2						
			1	138	69	2					
				0	68	34	2				
					1	34	17	2			
						0	16	8	2		
							1	8	4	2	
								0	4	2	2
									0	2	1
										0	

2216d ⟹ **0000.1000.1010.1000b (WORD)**

-2216d ⟹ 1) 0000.1000.1010.1000b — двоичный код числа |**-2216**|

2) 1111.0111.0101.0111b — инверсия

3) + 1

1111.0111.0101.1000b — **дополнительный код**

Таблица 1. Машинное представление заданных чисел.

Dec	Byte		Word	
	Bin	Hex	Bin	Hex
59	0011.1011	3B	0000.0000.0011.1011	003B
-59	**1100.0101**	**C5**	**1111.1111.1100.0101**	**FFC5**
21	0000.0101	15	0000.0000.0000.0101	0015
-21	**1110.1011**	**EB**	**1111.1111.1110.1011**	**FFEB**
2254			0000.1000.1100.1110	08CE
-2254			**1111.0111.0011.0010**	**F732**
2216			0000.1000.1010.1000	08A8
-2216			**1111.0111.0101.1000**	**F758**

Сделаем проверку, для этого напишем по аналогии с примером 2.10 программу, откомпилируем ее при помощи компилятора с языка *ассемблера TASM* и получим файл листинга.

Реализация в Turbo Assembler 3.2

Файл листинга Kontrol1.lst

```
Turbo Assembler  Version 3.2      15/10/00 11:33:54      Page 1
Kontrol1.asm
0000                         .MODEL tiny
0000                         .DATA
                              ; ********** Labor 1 **********

                              ;** Byte **
0000  3B                      lb      db      59
0001  15                              db      21

                              ;** Word **
0002  003B                    lw      dw      59
0004  0015                            dw      21
0006  08A8                            dw      2216
0008  08CE                            dw      2254

                              ;** ShortInt **
000A  C5                      lsh     db      -59
000B  EB                              db      -21

                              ;** Integer **
000C  FFC5                    li      dw      -59
000E  FFEB                            dw      -21
0010  F758                            dw      -2216
0012  F732                            dw      -2254

                              ;** LongInt **
0014  000008A8                ll      dd      2216
0018  000008CE                        dd      2254
001C  FFFFF732                        dd      -2254
0020  FFFFF758                        dd      -2216

                              END
```

Анализ этого файла показывает, что машинный формат целых чисел нами был получен правильно.



Лабораторная работа № 2

Внутреннее представление вещественных данных в IBM PC

Ничто в жизни не достигается без большого труда.

Квинт Гораций Флакк (65-8 гг. до н.э.)

Цель работы

Выполнить перевод заданных чисел из десятичной в двоичную систему счисления. Дать их внутреннее (машинное) представление в форматах типов **Single (float)**, **Double (double)**, **Extended (long double)**. Машинное представление данных должно быть дано в двоичной и шестнадцатеричной системах счисления.

Порядок работы

1) вычислить для своего варианта вещественные числа;
2) перевести их из 10-тичной в 2-ичную систему счисления и сделать нормализацию;
3) получить их внутреннее представление (числа указывает преподаватель);
4) написать программу описания этих чисел на языке Ассемблера и получить листинг;
5) проверить правильность своих выкладок.

В отчете по лабораторной работе должен быть представлен подробный протокол перевода всех заданных чисел из 10-тичной в 2-ичную систему счисления (**теоретическое введение**).

Варианты

Преподаватель задает базовое число ±X.Y. Студент должен добавить и отнять от него номер своего варианта (для целой и дробной части отдельно).

Например, пусть **X = ± 60, Y = ± 4567, № = 45**. Тогда получатся следующие двенадцать вещественных чисел для варианта **№ = 45**, а именно:

1) 60.4567
   + 45 45 = 105.4612;
2) 60.4567
   - 45 45 = 15.4522;
3) - 60.4567
   + 45 45 = -15.4522;
4) - 60.4567
   - 45 45 = -105.4612.

Остальные варианты — машинное представление отдельно дробной и целой части. Например,

5) 105;
6) 15;
7) -15;
8) -105;
9) 0.4612;
10) 0.4522;
11) -0.4522;
12) -0.4612.

Контрольные вопросы

1. С какими вещественными данными работает сопроцессор i8087?
2. Что такое характеристика?
3. Машинные форматы вещественных данных.
4. Диапазон допустимых значений для вещественных данных.
5. Что может означать, например, машинное представление 0F6549FDCh, и в каком формате?

При выполнении этой лабораторной работы мы воспользуемся материалом, изложенным в пп. 1.2.1-1.2.3 и 2.3, лабораторной работе № 1 и в прил.4.

Пример решения типового варианта

Вариант № 45. Базовые числа: X = ± 60, Y = ± 4567.

Таким образом, получаем для анализа все числа, перечисленные выше. Сделаем подробный протокол получения внутреннего представления для следующих чисел:

105.4612, **-105.** и **0.4612**

во всех трех вещественных форматах.

Решение

Типы переменных, их диапазоны значений и количество требуемой для них памяти приведены в следующей таблице:

Тип	Диапазон значений	Мантисса (кол-во цифр)	Требуемая память
Single (float)	1.5e-45...3.4e38	7-8	4 bytes
Double	5.0e-324...1.7e308	15-16	8 bytes
Extended (long double)	3.4e-4932...1.1e4932	19-20	10 bytes

ШАГ 1. Перевод десятичных чисел в двоичную систему счисления

Поскольку знак числа учитывается ТОЛЬКО в старшем бите внутреннего формата, то в шаге 1 и шаге 2 мы будем работать с ПОЛОЖИТЕЛЬНЫМИ числами.

Сначала переведем ЦЕЛОЕ число 105 в двоичную систему счисления (по аналогии с лабораторной работой № 1):

105	2					
104	52	2				
1	52	26	2			
	0	26	13	2		
		0	12	6	2	
			1	6	3	2
				0	2	1
					1	

105d → **1101001b**

Теперь переведем в двоичную систему счисления ПРАВИЛЬНУЮ десятичную дробь (см. примеры 1.3а-1.5а). Для проверки представления в 32-х битном формате нам достаточно 24-х бит в мантиссе (сделаем немного с запасом — 26 бит):

0.4612d → ??? bin

#бита	Бит	Мантисса (Dec)	Множитель	Пояснения
	0	4612		Исходное число
1	0	9224	2	Результаты умножения мантиссы на 2
2	1	8448		
3	1	6896		
4	1	3792		
5	0	7584		
6	1	5168		
7	1	0336		
8	0	0672		
9	0	1344		
10	0	2688		
11	0	5376		
12	1	0752		
13	0	1504		
14	0	3008		
15	0	6016		
16	1	2032		
17	0	4064		
18	0	8128		
19	1	6256		
20	1	2512		
21	0	5024		
22	1	0048		
23	0	0096		
24	0	0192		
25	0	0384		
26	0	0768		

0.4612d → **0.01110110000100010011010000b**

Теперь у нас есть вся информация для получения двоичного представления СМЕШАННОЙ дроби:

105.4612d → **1101001.01110110000100010011010000b**

ШАГ 2. Получение двоичного нормализованного представления

105d → 1101001b = **1.101001b *2⁶**

0.4612d → 0.01110110000100010011010000 =
1.110110000100010011010000b * 2⁻²

105.4612d → 1101001.01110110000100010011010000b =
1.101001.01110110000100010011010000b *2⁶

ШАГ 3. Формат Dword

На этом шаге нам надо вспомнить все, что было сказано в п. 2.3.2.1.

-105.0 (см. Пример 2.8а)

Характеристика = 7F + 6 = 85. Распишем теперь наше ОТРИЦАТЕЛЬНОЕ ВЕЩЕСТВЕННОЕ число (состоит только из ЦЕЛОЙ части) по битам:

1 1000 0101 **10100**1000000000000000
S ^
1

Теперь распишем по тетрадам, а затем в HEX-коде:

1100. 0010. 1**101.** **001**0. 0000. 0000. 0000. 0000
C 2 D 2 0 0 0 0

0.4612

Характеристика = 7F — 2 = 7D. Распишем теперь наше ПОЛОЖИТЕЛЬНОЕ число по битам:

0 0111 1101 **11011000010001001101 0000**
S ^
1

Теперь распишем по тетрадам, а затем в HEX-коде:

0 011. 1110. **1110. 1100. 0010. 0010. 0110. 1000. 0**
3 E E C 2 2 6 8

105.4612

Характеристика = 7F + 6 = 85. Распишем теперь нашу смешанную дробь по битам:

0 1000 0101 **101001011101100001000100110 10000**
S ^
1

Теперь распишем по тетрадам, а затем в HEX-коде:

0100. 0010. 1**101. 0010. 1110. 1100. 0010. 0010. 0110. 1000.0**
4 2 D 2 E C 2 2

ШАГ 4. Формат Qword

На этом шаге все действия аналогичны шагу 3, только разрядность и для самого числа, и для его составляющих — другая (см. п. 2.3.2.2 и Пример 2.7b).

-105.0

Характеристика = 3FF + 6 = 405. Распишем теперь наше ОТРИЦАТЕЛЬНОЕ ВЕЩЕСТВЕННОЕ число (состоит только из ЦЕЛОЙ части) по битам:

1 100 0000 0101 **101001**000000000000000
S ^
1

Теперь распишем по тетрадам, а затем в HEX-коде:

1100. 0000. 0101. **1010. 01**00. 0000. 0000. 0000. 0000. 0000. 0000. 0000. 0000. 0000. 0000. 0000
C 0 5 A 4 0 0 0 0 0 0 0 0 0 0 0

0.4612



Характеристика = 3FF — 2 = 3FD. Распишем теперь наше ПОЛОЖИТЕЛЬНОЕ число по битам:

0 011 1111 1101 **11011000010001001101000...**
S ^
1

Теперь распишем по тетрадам, а затем в HEX-коде:

0 011. 1111. 1101. **1101. 1000. 0100. 0100. 1101. 0000............**
3 F D D 8 4 4 D 0 ....

Получить ВСЕ разряды мантиссы мы НЕ сможем, т.к. нет информации. Однако для проверки правильности наших выкладок и такого представления достаточно.

105.4612

Характеристика = 3FF + 6 = 405. Распишем теперь нашу смешанную дробь по битам:

0 100 0000 0101 **1010 0101 1101 1000 0100 0100 1101 0000............**
S ^
1

Теперь распишем по тетрадам, а затем в HEX-коде:

0100. 0000. 0101. **1010. 0101. 1101. 1000. 0100. 0100. 1101. 0000.........**
4 0 5 A 5 D 8 4 4 D 0 ....

Опять получить ВСЕ разряды мантиссы мы НЕ сможем, т.к. нет информации. Но и того, что есть, вполне достаточно.

ШАГ 5. Формат Tbyte

На этом шаге все действия аналогичны шагу 4, только разрядность и для самого числа, и для его составляющих другая. Кроме того, **у мантиссы НЕТ скрытого разряда** — см. п. 2.3.2.3 и Пример 2.8с.

-105.0

Характеристика = 3FFF + 6 = 4005. Распишем теперь наше ОТРИЦАТЕЛЬНОЕ ВЕЩЕСТВЕННОЕ число (состоит только из ЦЕЛОЙ части) по битам:

1100 0000 0000 0101 **1101 0010** 0000 0000 0000 00
S

Теперь распишем по тетрадам, а затем в HEX-коде:

1100. 0000. 0000. 0101. **1101. 0010**. 0000. 0000. 0000. 0000. 0000. 0000. 0000. 0000. 0000. 0000...000
C 0 0 5 D 2 0 0 0 0 0 0 0 0 0 0 ...0

0.4612

Характеристика = 3FFF — 2 = 3FFD. Распишем теперь наше ПОЛОЖИТЕЛЬНОЕ число по битам:

0 011 1111 1111 1101 **1110 1100 0010 0010 0110 1000 0.........**
S

Теперь распишем по тетрадам, а затем в HEX-коде:

0 011. 1111. 1111. 1101. **1110. 1100. 0010. 0010. 0110. 1000. 0.........**
3 F F D E C 2 2 6 8 ....

Получить ВСЕ разряды мантиссы мы НЕ сможем, т.к. нет информации. Однако для проверки правильности наших выкладок и такого представления достаточно.

105.4612

Характеристика = 3FFF + 6 = 4005. Распишем теперь нашу смешанную дробь по битам:

0100 0000 0000 0101 **1101 0010 1110 1100 0010 0010 0110 1000 0.........**
S

Теперь распишем по тетрадам, а затем в HEX-коде:

0100. 0000. 0000. 0101. **1101. 0010. 1110. 1100. 0010. 0010. 0110. 1000. 0....**
4 0 0 5 D 2 E C 2 2 6 8 ....

Опять получить ВСЕ разряды мантиссы мы НЕ сможем, т.к. нет информации. Но и того, что есть, вполне достаточно.

Реализация в Microsoft Assembler 6.12

Файл листинга Asm_digt2.lst

```
Microsoft (R) Macro Assembler Version 6.12.7164    05/27/01 23:29:11
lab 2                                               Page 1 — 1
                              title lab 2
                              .MODEL  tiny
 0000                         .DATA
                              ; сопроцессор (FPU) i8087
                              ;———————— float (DWord)
 0000   C2D2EC22                   f      dd -105.4612
 0004   42D2EC22                          dd  105.4612
 0008   C1773C36                          dd  -15.4522
 000C   41773C36                          dd   15.4522
 0010   42D20000                          dd  105.
 0014   C2D20000                          dd -105.
 0018   41700000                          dd   15.
 001C   C1700000                          dd  -15.
 0020   3EEC2268                          dd  0.4612
 0024   BEEC2268                          dd -0.4612
 0028   3EE786C2                          dd  0.4522
 002C   BEE786C2                          dd -0.4522
                              ;———————— double (QWord)
 0030                              d      dQ -105.4612
        C05A5D844D013A93
 0038   405A5D844D013A93                  dQ  105.4612
 0040   C02EE786C226809D                  dQ  -15.4522
 0048   402EE786C226809D                  dQ   15.4522
 0050   405A400000000000                  dQ  105.
 0058   C05A400000000000                  dQ -105.
 0060   402E000000000000                  dQ   15.
 0068   C02E000000000000                  dQ  -15.
 0070   3FDD844D013A92A3                  dQ  0.4612
 0078   BFDD844D013A92A3                  dQ -0.4612
```

```
0080    3FDCF0D844D013A9            dQ  0.4522
0088    BFDCF0D844D013A9            dQ -0.4522
                        ; ———————— long double
0090                           t    dT -105.4612
        C005D2EC226809D49518
009A    4005D2EC226809D49518        dT  105.4612
00A4    C002F73C36113404EA4B        dT  -15.4522
00AE    4002F73C36113404EA4B        dT   15.4522
00B8    4005D200000000000000        dT  105.
00C2    C005D200000000000000        dT -105.
00CC    4002F000000000000000        dT   15.
00D6    C002F000000000000000        dT  -15.
00E0    3FFDEC226809D495182B        dT  0.4612
00EA    BFFDEC226809D495182B        dT -0.4612
00F4    3FFDE786C226809D4952        dT  0.4522
00FE    BFFDE786C226809D4952        dT -0.4522
                        END         ; конец asm-модуля
```

Анализ этого файла показывает, что машинный формат вещественных чисел нами был получен правильно.

Реализация этой задачи в TASM даст аналогичные результаты, но для некоторых чисел с дробной частью в форматах 64 и 80 бит за счет округления получаются разные результаты в младшей HEX—цифре (эти числа в листинге выделены). Кроме того, 16-разрядный TASM (tasm.exe) НЕ поддерживает длинных имен (**Asm_digt2**), т.е. имен с количеством символов более чем восемь. 32-разрядный TASM (tasm32.exe) компилирует с тем же результатом, что и tasm.exe, но с поддержкой длинных имен — см. соответствующий листинг в материале, который прилагается к книге (а еще лучше — сделай сам!).

3

Лабораторная работа № 3

Вычисление целочисленных арифметических выражений (процессор i8086/i286)

Без примеров невозможно ни правильно учить, ни успешно учиться.

Колумелла Луций Юний Модерат (I в. н. э.)

Цель работы

Вычислить заданное целочисленное выражение для исходных данных в знаковых форматах длиной 8 и 16 бит: **ShortInt (signed char)** и **Integer (int)**, используя арифметические операции ADD, ADC, INC, SUB, SBB, DEC, NEG, IMUL, IDIV, CBW, CWD и, если нужно, логические операции SAL, SAR. Исходные данные должны вводиться с проверкой корректности вводимых символов. Формат результата зависит от специфики решаемого выражения. Входные данные и результат должны быть проверены на область допустимых значений.

Порядок работы

1) внимательно изучить свой вариант арифметического выражения;
2) написать модуль вычисления арифметического выражения на языке Ассемблера;
3) написать на базовом алгоритмическом языке программу корректного ввода исходных данных (с контролем допустимого диапазона), вычисления арифметического выражения и вывода полученного результата;
4) встроить вызов этого модуля в программу на базовом алгоритмическом языке;
5) произвести тестовые проверки, отметить нормальные и аномальные результаты, сделать анализ результатов.

Варианты

1) (2*c — d + 23) /(a/4 — 1);
2) (c + 4*d — 123)/(1 — a/2);
3) (-2*c + d*82)/(a/4 — 1);
4) (2*c + d — 52)/(a/4 + 1);
5) (c/4 — d*62)/(a*a + 1);
6) (-2*c — d + 53)/(a/4 — 1);
7) (2*c — d/4)/(a*a + 1);
8) (2 + c — d*23)/(2*a*a — 1);
9) (2*c — d/3)/(b — a/4);
10) (4*c + d — 1)/(c — a/2);
11) (2*c — d*42)/(c + a — 1);
12) (25/c — d + 2)/(b + a*a-1);
13) (c — d/2 + 33)/(2*a*a-1);
14) (4*c — d/2 + 23)/(a*a — 1);
15) (c*d + 23)/(a/2 -4*d- 1);
16) (c/d + 3*a/2)/(c — a + 1);
17) (2*c + d*51)/(d — a — 1);
18) (2*c + d/4 + 23)/(a*a — 1);
19) (2*c — d/2 +1)/(a*a+7);
20) (2*c/d + 2)/(d — a*a — 1);
21) (12/c — d*4 +73)/(a*a+1);
22) (2*c/a — d*d)/(d + a — 1);
23) (-53/a + d -4*a)/(1+a*b);
24) (-15*a + b — a/4)/(b*a -1);
25) (-25/a +c-b*a)/(1+c*b/2);
26) (4*a — 1 + b/2)/(b*c — 5);
27) (8*b + 1 — c)/(a/2+ b*c);
28) (4*a — b — 1)/(c/b + a);
29) (4*b/c — 1)/(12*c+a — b);
30) (b + c*b — a/4)/(a*b — 1);
31) (a + c/b — 28)/(4*b*a+1);
2) (c/b — 24 + a)/(2*a*c — 1);
33) (2*b — a + b*c)/(c/4- 1);
34) (41 — d/4 — 1)/(c/b + a*d);
35) (a — b*4 — 1)/(c/31+a*b);
36) (b/a + 4*c)/(c — b + 1);
37) (21 — a*c/4)/(1+c/a + b);
38) (c — 33 + b/4)/(a*c/b — 1);
39) (2*b — 38*c)/(b+a/c + 1);
40) (c/4 + 28*d)/(a/d — c — 1);
41) (a*b/4 — 1)/(41-b*a + c);
42) (1 + 6*a — b/2)/(c + a/d);
43) (a*b + 2*c)/(41-b/c + 1);
44) (4*b — c*a)/(b + c/28 — 1);
45) (2*c + a — 21)/(c/a*b+1);
46) (4/c + 3*a)/(c/a — b — 1);
47) (8*b — 1 — c)/(a*2+ b/c);
48) (4*a/b + 1)/(c*b — 18 + a);
49) (4*b/c + 1)/(2*c+a*c- b);
50) (b — c/b + a/4)/(a*b — 1);
51) (a — c*b + 28)/(4*b/a+1);
52) (c*b — 24 + a)/(b/2*c — 1);
53) (2*b — a + b/c)/(c/4+ 1);
54) (41*d/4 + 1)/(a-c/b + a*d);
55) (a + b/4 — 1)/(c/3-a*b);
56) (b*a + c/2)/(4*c — b + 1);
57) (25 + 2*a/c)/(c*a-b — 1);
58) (c + 23 — b*4)/(a*c/b — 1);
59) (b/2 — 53/c)/(b-a*c + 1);
60) (c*4 + 28/d)/(a*d — c — 1);

Контрольные вопросы

1. Особенности выполнения изучаемых базовых команд процессора i8086/i286.
2. Понятие о байтах кода операции, способах адресации, регистрах и смещениях.
3. Ассемблирование и дизассемблирование команд на своих примерах.
4. Необходимость в контроле диапазона целочисленных данных при вводе.
5. Понятие о 16- и 32-разрядных вычислительных платформах.
6. Диапазон допустимых значений для целочисленных переменных.

Пример решения типового варианта

Вариант № 62.

Вычислить заданное целочисленное выражение:

(25/c — d + 2)/(d + a*a-1);

для исходных данных в знаковых форматах длиной 8 и 16 бит: **ShortInt (signed char)** и **Integer (int)**, используя арифметические операции ADD, ADC, INC, SUB, SBB, DEC, NEG, IMUL, IDIV, CBW, CWD и, если нужно, логические операции SAL, SAR.

Решение

Сделаем этот пример в трех вариантах реализации с привлечением разных алгоритмических языков и сред программирования. Предложенные вашему вниманию главные программы на алгоритмических языках Паскаль и C++, а также ассемблерные модули демонстрируют ПРИНЦИПЫ решения ДАННОЙ задачи. В целях *наглядности и простоты изложения* автор СОЗНАТЕЛЬНО НЕ делала ОПТИМИЗАЦИЮ ПРОГРАММ, поскольку бывают ситуации, когда оптимизация может и напортить. *Легче сделать корректную программу быстрой, чем быструю программу корректной. Самым плохим временем для оптимизации является момент, когда вы начинаете писать код* [Л7-9]. У программистов даже есть такое правило: "лучшее — враг хорошего".

У вашего варианта, вполне вероятно, будут другие, свои, СПЕЦИФИЧЕСКИЕ особенности, отличные от тех, которые мы рассматриваем. Охватить их все в рамках ОДНОЙ книги немыслимо. Но, на что надо обращать внимание, здесь и в других местах книги я буду стараться указывать. **Учитесь эти особенности ВИДЕТЬ**. Это, в конечном счете, и есть САМОЕ ГЛАВНОЕ. Для этого нужен опыт, а для приобретения опыта надо САМОМУ решать задачи. И, желательно, правильно. Другого пути нет! Итак, в добрый и плодотворный путь.

При выполнении данной лабораторной работы нам понадобятся в основном материалы глав 5-6 и, конечно, п. 2.2. Проиллюстрируем наши действия по шагам.

Вариант 1. Реализация в Borland Pascal 7.x и Turbo Assembler 3.2

Чтобы ГРАМОТНО запрограммировать решение этой задачи, начнем с Ассемблера, воспользовавшись идеями примера 5.4b.

ШАГ 1.

Процедура вычисления арифметического выражения для 8-битных данных — Lab3S

В качестве исходных данных пусть будут переменные **aS (a)**, **ccS(c)** и **ddS(d)**:

```
.data
Extrn    aS:Byte,ccS:Byte,dds:Byte
```

Вычисления начнем со знаменателя (**Denominator**), поскольку тон в вычислениях нашего арифметического выражения задает именно он. Затем вычислим числитель (**Numerator**). Из-за того, что в знаменателе стоит операция УМНОЖЕНИЯ **a*a**, 8-битное данное **d (ddS)** придется расширить до слова (16 бит), чтобы не потерять СТАРШУЮ часть ПРОИЗВЕДЕНИЯ. Таким образом, знаменатель получается длиной 16 бит,

значит, числитель должен быть 32-битный. Результат вычисления арифметического выражения (xI) соответственно будет тоже 16-битный. Значения числителя и знаменателя передадим для последующей визуализации (и проверки) в Паскаль. Получаем такой модуль на Ассемблере:

```
                  title Lab3asm.asm
;CopyRight by Голубь Н.Г., 1993-1997, 2001
          .MODEL Large, Pascal
; Компиляция: tasm asmfile.asm /l
; Вычислить заданное целочисленное выражение для
; исходных данных в знаковых форматах длиной
;         8 и 16 бит (shortInt и integer):
;    Вариант # 62) (25/c - d + 2)/(d + a*a-1);
          .data
          Extrn   Numerator:Dword
          Extrn   aS:Byte,ccS:Byte,dds:Byte
          Extrn   xI:Word
          Extrn   Denominator:Word
          .code
          Public  Lab3S
Lab3S   proc    far
;знаменатель - Denominator ──────────────
          mov     aL,aS
          Imul    aL          ; <ax>=a*a
          dec     ax          ; <ax>=a*a-1
          ; подготовка к расширению d (BYTE===>WORD)
; Запоминание <ax>=a*a-1 в стеке
          push    ax
          mov     aL,dds
          CBW                 ; aL ===> AX
; <SI>=<ax>=d - нужно для ЧИСЛИТЕЛЯ!!!
          mov     SI,ax
; Восстановление содержимого стека ===> <bx>=a*a-1
          pop     bx
          add     bx,ax       ; <bx>=a*a-1+d
          mov     Denominator,bx
;числитель - Numerator ────────────────
          mov     ax, 25
          IDIV    ccS         ; <aL>=25/c
          CBW                 ; aL ===> AX
          sub     ax,SI       ; <ax>=25/c-d
          add     ax,2        ; <ax>=25/c-d+2
          CWD                 ; aX ===> DX:AX
          mov     WORD PTR Numerator, AX    ; мл. часть
          mov     WORD PTR Numerator+2, DX ; ст. часть
; РЕЗУЛЬТАТ
          IDIV    bx          ; <ax>=(25/c-d+2)/(a*a-1+d)
          mov     xI,ax
          ret
Lab3S   endp
          Public  Lab3I
```

```
. Lab3I    proc     far
           ; Пока  НИЧЕГО  НЕТ!!!!
            mov      xI,ax          ;  xI <=== <ax>
            ret
Lab3I      endp
            end
```

ШАГ 2.

Программа для проверки работы процедуры Lab3S

Теперь нам понятно, на что надо обратить внимание на Паскале, не правда ли? Поэтому на Паскале функция вычисления нашего арифметического выражения будет иметь следующий вид: **function F (a,c,d:LongInt; Min,Max:Longint) : Boolean;**

Организуем корректный ввод исходных данных и вызов процедур на Ассемблере (хотя готова пока только одна процедура, а вместо другой мы сделали заглушку — пустое тело процедуры).

Исходный текст программы Lab3P.pas

```
Program Lab3P;
{———————————————————————————————————————————
              Лабораторная работа # 3.
Вычислить заданное целочисленное выражение для исходных данных
  в знаковых форматах длиной 8 и 16 бит: signed char и int:
     Вариант # 62) (25/c — d + 2)/(d + a*a-1);
        Корректный ввод числовых символов.
          Вариант С КОНТРОЛЕМ данных
      CopyRight by Голубь Н.Г., 1993-1997,2001
———————————————————————————————————————————}
{$I Lab3asm}    {вызов ASM-модуля Lab3asm.obj}
{$f+}
Uses CRT, InputNum;
Const
  {Символьные константы}
  inv2='Результат ';
  invAi='A (Integer)';
  invCi='C (Integer)';
  invDi='D (Integer)';
  invAs='A (ShortInt)';
  invCs='C (ShortInt)';
  invDs='D (ShortInt)';

  {Допустимый диапазон для вводимых данных и результата}
  IntMin=-32768;
  IntMax= 32767;
  ShortMin=-128;
  ShortMax= 127;

Label L1;
var  {Описание ГЛОБАЛЬНЫХ переменных}
  a,c,d,x,Numerator                : LongInt;
  aS,ccS,ddS                       : ShortInt;
  xI,aI,cI,ddI,Denominator: Integer;
  ch                               : Char;
```

```
{Внешние процедуры на Ассемблере}
Procedure Lab3I(var x:Integer; aI,cI,ddI:Integer); external;
Procedure Lab3S(var x:Integer; aS,ccS,ddS:ShortInt); external;

{Функция вычисления x=(25/c — d + 2)/(d + a*a-1)c проверкой}
{РЕЗУЛЬТАТА на допустимый диапазон}
functión F (a,c,d:LongInt; Min,Max:Longint) : Boolean;
Var x1 : LongInt;
    s1 : String;
Begin
  F:=True;  {Ошибок НЕТ}
  x1:=d + a*a-1;
{Проверка выхода за допустимый диапазон ЗНАМЕНАТЕЛЯ}
  if (x1<Min)OR(x1>Max) then
       Begin
         s1:='Знаменатель '+inv3;  {Формирование сообщения об ошибке}
         Writeln(s1,Min,'..',Max,']!!!!');
         {Вывод полученного значения, выходящего ЗА ДОПУСТИМЫЙ диапазон}
         Writeln(x1);
         Writeln(inv1);
         F:=False; {Признак ошибки}
         Exit;
       End;
  if ((c=0) OR(x1=0)) then
    begin
      Writeln('Ошибка!!! Деление на НОЛЬ!!!');
      F:=False; {Признак ошибки}
    end
  else
    begin
      x:=25 div c — d + 2;
      Writeln('Pascal:');
      Writeln('Числитель    ',x);
      Writeln('Знаменатель  ',x1);
      x:=x div x1;
      Writeln('          x=',x);
{Проверка выхода за допустимый диапазон РЕЗУЛЬТАТА}
      if (x<Min)OR(x>Max) then
        Begin
         s1:=inv2+inv3;  {Формирование сообщения об ошибке}
         Writeln(s1,Min,'..',Max,']!!!!');
        {Вывод полученного значения, выходящего ЗА ДОПУСТИМЫЙ диапазон}
         Writeln(x);
         Writeln(inv1);
         F:=False; {Признак ошибки}
         Exit;
        End
    end
End;

{—— Блок процедур работы с ОПРЕДЕЛЕННЫМ типом данных ——}
Procedure ExpI;  {Integer}
Label L1;
Begin
  Writeln('=============== Integer ====================');
```

```
  L1:
    InputNumber(a,invAi,IntMin,IntMax);
    InputNumber(c,invCi,IntMin,IntMax);
    InputNumber(d,invDi,IntMin,IntMax);
    if NOT(F(a,c,d,IntMin,IntMax)) then {если есть ошибки}
      goto L1; {повторяем ввод данных}
    aI:=a;cI:=c;ddI:=d;
    Lab3I(xI,aI,cI,ddI);
    writeln ('АССЕМБЛЕР:');
    Writeln('Числитель    ',Numerator);
   Writeln('Знаменатель  ',Denominator);
    Writeln('          x= ',xI);
End;

Procedure ExpS;{ShortInt}
Label L1;
Begin
  Writeln('================== ShortInt =====================');
  L1:
    InputNumber(a,invAs,ShortMin,ShortMax);
    InputNumber(c,invCs,ShortMin,ShortMax);
    InputNumber(d,invDs,ShortMin,ShortMax);
    if NOT (F(a,c,d,IntMin,IntMax)) then   {если есть ошибки}
     goto L1; {повторяем ввод данных}
    aS:=a;ccS:=c;ddS:=d;
    Lab3S(xI,aS,ccS,ddS);
    writeln ('АССЕМБЛЕР:');
    Writeln('Числитель    ',Numerator);
   Writeln('Знаменатель  ',Denominator);
    Writeln('          x= ',xI);
End;
{-----------------------------------------------------------------}
Procedure Main;
Var i :Integer;
Begin
  Writeln(' Вычислить:      x = (25/c — d + 2)/(d + a*a-1);');
  Repeat
    Writeln('Введите Ваш выбор параметров x,a,b');
    Writeln('      1 — Integer');
    Writeln('      2 — ShortInt');
    Writeln('      3 — Выход');
  {Контроль ввода НЕ числовых символов}
  {$I-}
    Readln(i);     {Промежуточный буфер ввода для ЦЕЛОЧИСЛЕННЫХ данных}
  {$I+}
  Until (IOResult=0);
  case i of
    1: ExpI;
    2: ExpS;
    3: HALT(0); {Нормальный выход из программы с кодом 0}
  else Exit;
  end
End;
begin
```

```
  Repeat
    ClrScr;
    Main;
    Writeln('Повторим? (y/n)');
    ch:=ReadKey;
  Until (ch='n') or (ch='N');
end.
```

ШАГ 3.
Тестирование реализации для 8-битных исходных данных

```
 Вычислить:        x = (25/c — d + 2)/(d + a*a-1);
Введите Ваш выбор параметров x,a,b
       1 — Integer
       2 — ShortInt
       3 — Выход
2
================== ShortInt =====================
Введите значение A (ShortInt)===>0
Введите значение C (ShortInt)===>0
Введите значение D (ShortInt)===>0
Ошибка!!! Деление на НОЛЬ!!!
Введите значение A (ShortInt)===>0
Введите значение C (ShortInt)===>1
Введите значение D (ShortInt)===>1
Ошибка!!! Деление на НОЛЬ!!!
Введите значение A (ShortInt)===>0
Введите значение C (ShortInt)===>1
Введите значение D (ShortInt)===>0
Pascal:
Числитель     27
Знаменатель   -1
              x=-27
АССЕМБЛЕР:
Числитель     27
Знаменатель   -1
             x= -27
Повторим? (y/n)
 Вычислить:        x = (25/c — d + 2)/(d + a*a-1);
Введите Ваш выбор параметров x,a,b
       1 — Integer
       2 — ShortInt
       3 — Выход
2
================== ShortInt =====================
Введите значение A (ShortInt)===>127
Введите значение C (ShortInt)===>-1
Введите значение D (ShortInt)===>-128
Pascal:
Числитель     105
Знаменатель   16000
              x=0
АССЕМБЛЕР:
Числитель     105
```

```
Знаменатель   16000
             x= 0
Повторим? (y/n)
 Вычислить:         x = (25/c - d + 2)/(d + a*a-1);
Введите Ваш выбор параметров x,a,b
       1 - Integer
       2 - ShortInt
       3 - Выход
2
================= ShortInt =====================
Введите значение A (ShortInt)===>0
Введите значение C (ShortInt)===>-1
Введите значение D (ShortInt)===>127
Pascal:
Числитель      -150
Знаменатель   126
              x=-1
АССЕМБЛЕР:
Числитель      -150
Знаменатель   126
             x= -1
Повторим? (y/n)
 Вычислить:         x = (25/c - d + 2)/(d + a*a-1);
Введите Ваш выбор параметров x,a,b
       1 - Integer
       2 - ShortInt
       3 - Выход
2
================= ShortInt =====================
Введите значение A (ShortInt)===>-2
Введите значение C (ShortInt)===>-1
Введите значение D (ShortInt)===>-128
Pascal:
Числитель      105
Знаменатель   -125
              x=0
АССЕМБЛЕР:
Числитель      105
Знаменатель   -125
             x= 0
Повторим? (y/n)
```

Итак, по результатам тестирования Ассемблер и Паскаль работают НОРМАЛЬНО.

ШАГ 4.

Процедура вычисления арифметического выражения для 16-битных данных — Lab3I

Теперь можно заняться и 16-битными исходными данными. Полной аналогии с реализацией для 8-битных данных НЕ получится, поскольку мы НЕ умеем пока делить на 32-разрядные данные. Поэтому при вычислении функции на Паскале мы и поставили проверку диапазона для знаменателя (для 8-битных данных она в принципе была НЕ нужна!). При вычислении 32-битного числителя нам придется повозиться, потому что сначала нужно будет сделать операцию деления для знаковых 16-битных

данных **25/c**, а затем 16-битный результат этой операции (в регистре **AX**) надо будет расширить до 32-битного. И дальнейшие операции вычитания и сложения делать уже с 32-битными данными. Иначе придется на диапазон числителя накладывать ограничение в Паскале, что резко сузит диапазон вводимых данных (область определения нашей функции). Это еще раз подтверждает правильность выбора нами типа исходных данных в Паскале как **LongInt**.

В результате получаем следующую процедуру на Ассемблере:

```
            Public   Lab3I
Lab3I    proc     far
;знаменатель — Denominator ——————————————
         mov      aX,aI
         Imul     aI          ; <dx:ax>=a*a
         dec      ax          ; <ax>=a*a-1
         add      ax,ddI      ; <ax>=a*a-1+d
         mov      Denominator,ax
;числитель — Numerator ——————————————
         mov      ax, 25
         CWD                  ; aX ===> DX:AX
         IDIV     cI          ; <ax>=25/c
         CWD                  ; aX ===> DX:AX
; Запоминание <dx> в стеке (25/c — ст. часть)
         push     dx
; Запоминание <ax>=25/c в стеке (мл. часть)
         push     ax
         mov      ax,ddI
         CWD                  ; aX ===> DX:AX
; Восстановление содержимого стека ===> <bx>=25/c — мл. часть
         pop      bx
; Восстановление содержимого стека ===> <cx>=25/c — ст. часть
         pop      cx
         sub      bx,ax       ; <bx>=25/c-d — мл. часть
         sbb      cx,dx       ; <cx>=25/c-d — ст. часть
         add      bx,2        ; <bx>=25/c-d+2
         adc      cx,0
; мл. часть числителя
         mov      WORD PTR Numerator, BX
; ст. часть числителя
         mov      WORD PTR Numerator+2, CX ;
         ; готовимся к делению
         mov      ax,bx
         mov      dx,cx
; РЕЗУЛЬТАТ <ax>=(25/c-d+2)/(a*a-1+d)
         IDIV     Denominator
         mov      xI,ax
         ret
Lab3I    endp
```

Осталось немного изменить описание внешних данных:

```
Extrn    xI:Word,aI:Word,cI:Word,ddI:Word
```

Модуль на Ассемблере **Lab3asm.asm** готов. Осталось протестировать нашу процедуру.

ШАГ 5.
Тестирование реализации для 16-битных исходных данных

```
 Вычислить:        x = (25/c - d + 2)/(d + a*a-1);
Введите Ваш выбор параметров x,a,b
       1 - Integer
       2 - ShortInt
       3 - Выход
1
=============== Integer ======================
Введите значение A (Integer)===>-2222
Введите значение C (Integer)===>-1
Введите значение D (Integer)===>22222
Знаменатель выходит за диапазон [-32768..32767]!!!!
4959505
Повторите ввод.
Введите значение A (Integer)===>1
Введите значение C (Integer)===>-1
Введите значение D (Integer)===>32767
Pascal:
Числитель     -32790
Знаменатель   32767
              x=-1
АССЕМБЛЕР:
Числитель     -32790
Знаменатель   32767
             x= -1
Повторим? (y/n)
 Вычислить:        x = (25/c - d + 2)/(d + a*a-1);
Введите Ваш выбор параметров x,a,b
       1 - Integer
       2 - ShortInt
       3 - Выход
1
=============== Integer ======================
Введите значение A (Integer)===>111
Введите значение C (Integer)===>-25
Введите значение D (Integer)===>-32768
Pascal:
Числитель      32769
Знаменатель   -20448
              x=-1
АССЕМБЛЕР:
Числитель      32769
Знаменатель   -20448
             x= -1
Повторим? (y/n)
 Вычислить:        x = (25/c - d + 2)/(d + a*a-1);
Введите Ваш выбор параметров x,a,b
       1 - Integer
       2 - ShortInt
       3 - Выход
1
=============== Integer ======================
Введите значение A (Integer)===>0
```

```
Введите значение C (Integer)===>1
Введите значение D (Integer)===>1
Ошибка!!! Деление на НОЛЬ!!!
Введите значение A (Integer)===>200
Введите значение C (Integer)===>1
Введите значение D (Integer)===>-32222
Pascal:
Числитель     32249
Знаменатель   7777
              x=4
АССЕМБЛЕР:
Числитель     32249
Знаменатель   7777
            x= 4
Повторим? (y/n)
 Вычислить:         x = (25/c – d + 2)/(d + a*a-1);
Введите Ваш выбор параметров x,a,b
      1 – Integer
      2 – ShortInt
      3 – Выход
1
=============== Integer =====================
Введите значение A (Integer)===>100
Введите значение C (Integer)===>1
Введите значение D (Integer)===>-9998
Pascal:
Числитель     10025
Знаменатель   1
              x=10025
АССЕМБЛЕР:
Числитель     10025
Знаменатель   1
            x= 10025
Повторим? (y/n)
 Вычислить:         x = (25/c – d + 2)/(d + a*a-1);
Введите Ваш выбор параметров x,a,b
      1 – Integer
      2 – ShortInt
      3 – Выход
1
=============== Integer =====================
Введите значение A (Integer)===>181
Введите значение C (Integer)===>1
Введите значение D (Integer)===>-32759
Pascal:
Числитель     32786
Знаменатель   1
              x=32786
Результат выходит за диапазон [-32768..32767]!!!!
32786
Повторите ввод.
Введите значение A (Integer)===>181
Введите значение C (Integer)===>-1
Введите значение D (Integer)===>-32759
Pascal:
```

```
Числитель      32736
Знаменатель    1
               x=32736
АССЕМБЛЕР:
Числитель      32736
Знаменатель    1
              x= 32736
Повторим? (y/n)
```

Тесты показывают, что все считается вроде бы верно. Проверьте и вы на ДРУГИХ данных, воспользовавшись рекомендациями п.6.3.



ЗАМЕЧАНИЕ.

При стыковке модулей на Ассемблере с программами на алгоритмическом языке Паскаль с использованием среды программирования **Borland Pascal 7.01** можно брать и компиляторы **tasm.exe** версий выше, чем 3.2 (но НЕ ниже!). Наверное, исследуя листинги в прилагаемых к данной книге материалах, вы уже обратили на это внимание.

Вариант 2. Реализация в Borland C++ 3.1 и Turbo Assembler 3.1

Решение этой задачи в среде Borland C++ 3.1 от решения в среде Borland Pascal 7.01 АЛГОРИТМИЧЕСКИ практически НИЧЕМ отличаться НЕ будет. Конечно, изменится *интерфейс стыковки* Cpp+ASM. Сам же код на Ассемблере останется тот же.

При реализации программы на C++ воспользуемся идеями примеров 6.2a и 6.2b. Целиком воспользоваться идеями примера 6.2b, к сожалению, НЕ получится, т.к. Borland C++ 3.1 не поддерживает динамического определения типа данных. Шаблоны, однако, он уже поддерживает. В результате получим следующие два файла.

Исходный текст программы Lab3c.cpp

```
/* Lab3c.cpp              Borland C++ 3.1
                     Проект  DOS (standard)
                          asm+cpp файлы
                     (c) Copyright 1998-2001 by Голубь Н.Г.
                      Лабораторная работа # 3.
Вычислить заданное целочисленное выражение для исходных данных
в знаковых форматах длиной 8 и 16 бит: signed char и int:
      Вариант # 62) (25/c — d + 2)/(d + a*a-1);
          Корректный ввод числовых символов.
*/

#include <iostream.h>
#include <conio.h>
#include "inputNum.h" // Ввод ЛЮБЫХ ЦЕЛЫХ чисел
typedef unsigned char byte;
const char* InputA = " ???? a ===> ";
const char* InputC = " ???? c ===> ";
const char* InputD = " ???? d ===> ";
const char* ResultC =     "\n\n Result C++ = ";
const char* ResultASM =   "\n\n Result ASM = ";
const char* NUMERATOR   = "\n ЧИСЛИТЕЛЬ    = ";
const char* DENOMINATOR = "\n ЗНАМЕНАТЕЛЬ  = ";
const char* CONTINUE = "\n\n Нажмите любую клавишу для продолжения...";
const char* CALC = "\n\t  Вычислить вариант # 62) (25/c — d + 2)/(d + a*a-1);\n";
```

```
const char* SELECT = "\n Выберите тип аргументов:\n\n";
const char* CHAR =   "\t 1 — char\n";
const char* INT =    "\t 2 — int\n";
const char* EXIT =   "\t 3 — Выход из этой программы...\n";
const char* RESULT = "\n!!!!!!! Результат ";
const char* RANGE = " выходит за диапазон int ";
const char* DIVIDE = "\nОшибка!!!!! Деление на ноль !!!\n";
const char* ARGUMENT = "\n Тип аргументов ";

// ВНЕШНИЕ ассемблерные функции
extern "C"
{
 void Lab3S(void);
 void Lab3I(void);
}
// Контроль диапазона для 16-ти разрядных знаковых чисел:
 //              -32768<=a<=32767
 //(c) Copyright 1998 by Артем Астафьев (629 группа ХАИ)
inline  int    test(long int a)
    {return ((a>>15)+1)&~1;}

void title(void)
{
 cout << CALC;
}

void title(long int a, long int c, long int d)
{
// (25/c — d + 2)/(d + a*a-1)
 cout << "\n          (25/"<< c << " — " << d << " + 2)"
       << "\n  ——————————————————————————————————————— "
       << "\n         (" << d << " + " << a << "*" << a << " — 1)\n";
}

void PressAnyKey()
{ cout <<  CONTINUE;
   getch();
}
char menu(void)
{
 cout << SELECT
       << CHAR
       << INT
       << EXIT;
 return getch();
}

//——————————— ГЛОБАЛЬНЫЕ переменные — передаются в Ассемблер
long Numerator;
char aS, ccS, dds;
int  Denominator, xI, aI, cI, ddI;
int t=1;

//———————————————————————————————————————————————
```

```
// ПАРАМЕТРИЗОВАННАЯ функция вычисления арифметического выражения
template <class Type>
int calculate(Type a, Type c, Type d)
{
   long res;
   long dn = d + (long)a * a — 1;
   if (test(dn))
      { // Ошибка диапазона для переменной типа int — ЗНАМЕНАТЕЛЬ!!!
     cout << DENOMINATOR << dn << RANGE;
     return 0;
      }
   if (!c||!dn)
     {
     cout << DIVIDE;
     return 0; // Деление на ноль!!!
      }
   else
    {
     long n = 25/c — (long)d + 2;
     res = n/dn;
     cout << ResultC << res;
     cout << NUMERATOR << n;
     cout << DENOMINATOR << dn;
     if (test(res))
      { // Ошибка диапазона для переменной типа int — РЕЗУЛЬТАТ!!!
       cout << RESULT << RANGE;
       return 0;
      }
     }
  return 1;  // Ok!
}

void ProcChar()
{
 cout << ARGUMENT << " CHAR (-128..127) \n\n";
 while (input (InputA,aS));
 while (input (InputC,ccS));
 while (input (InputD,dds));
 title(aS,ccS,dds);
 if (calculate(aS,ccS,dds))  // Ok!
  { Lab3S();
    cout << ResultASM << xl;
    cout << NUMERATOR << Numerator;
    cout << DENOMINATOR << Denominator;
  }
}

void ProcInt()
{
 cout << ARGUMENT << " INT (-32768..32767)\n\n";
 while (input (InputA,al));
 while (input (InputC,cl));
 while (input (InputD,ddl));
 title(al,cl,ddl);
 if (calculate(al,cl,ddl))  // Ok!
```

```
  { Lab3I();
    cout << ResultASM << xI;
    cout << NUMERATOR << Numerator;
    cout << DENOMINATOR << Denominator;
  }
}

int main(void)
{
 char point;
 do{
    clrscr();
    title();
    point = menu();
    switch(point)
     {
     case '1':ProcChar();break;
     case '2':ProcInt(); break;
     case '3': return 0;
    }
    cout << "\n==================== test #" << t++ << " =================\n";
    PressAnyKey();
  }while (point != '3');
}
```

Исходный текст модуля Lab3asm.asm

```
             title Lab3asm.asm
;CopyRight by Голубь Н.Г., 1993-1997, 2001
            .MODEL Large, C
; Компиляция: tasm asmfile.asm /l /ml
; или в среде BC++3.1
; Вычислить заданное целочисленное выражение для
; исходных данных в знаковых форматах
; длиной 8 и 16 бит (shortInt и integer):
;    Вариант # 62) (25/c - d + 2)/(d + a*a-1);
            .data
  Extrn     C Numerator:Dword
  Extrn     C aS:Byte,ccS:Byte,dds:Byte
  Extrn     C xI:Word,aI:Word,cI:Word,ddI:Word
  Extrn     C Denominator:Word
            .code
            Public   C Lab3S
Lab3S    proc     far
;знаменатель - Denominator --------------
            mov        aL,aS
            Imul       aL              ; <ax>=a*a
            dec        ax              ; <ax>=a*a-1
            ; подготовка к расширению d (BYTE===>WORD)
; Запоминание <ax>=a*a-1 в стеке
            push       ax
            mov        aL,dds
            CBW                        ; aL ===> AX
```

```
; <SI>=<ax>=d - нужно для ЧИСЛИТЕЛЯ!!!
          mov      SI,ax
; Восстановление содержимого стека ===> <bx>=a*a-1
          pop      bx
          add      bx,ax        ; <bx>=a*a-1+d
          mov      Denominator,bx
;числитель - Numerator ---------------
          mov      ax, 25
          IDIV     ccS          ; <aL>=25/c
          CBW                    ; aL ===> AX
          sub      ax,SI        ; <ax>=25/c-d
          add      ax,2         ; <ax>=25/c-d+2
          CWD                    ; aX ===> DX:AX
; мл. часть числителя
          mov      WORD PTR Numerator, AX
; ст. часть числителя
          mov      WORD PTR Numerator+2, DX
; РЕЗУЛЬТАТ
          IDIV     bx           ; <ax>=(25/c-d+2)/(a*a-1+d)
          mov      xI,ax
          ret
Lab3S    endp
;=============================================
          Public   C Lab3I
Lab3I    proc     far
;знаменатель - Denominator -------------
          mov      aX,aI
          Imul     aI           ; <dx:ax>=a*a
          dec      ax            ; <ax>=a*a-1
          add      ax,ddI       ; <ax>=a*a-1+d
          mov      Denominator,ax
;числитель - Numerator ---------------
          mov      ax, 25
          CWD                    ; aX ===> DX:AX
          IDIV     cI           ; <ax>=25/c
          CWD                    ; aX ===> DX:AX
; Запоминание <dx> в стеке (25/c - ст. часть)
          push     dx
; Запоминание <ax>=25/c в стеке (мл. часть)
          push     ax
          mov      ax,ddI
          CWD                    ; aX ===> DX:AX
; Восстановление содержимого стека ===> <bx>=25/c
;   - мл. часть
          pop      bx
; Восстановление содержимого стека ===> <cx>=25/c
;   - ст. часть
          pop      cx
          sub      bx,ax        ; <bx>=25/c-d - мл. часть
          sbb      cx,dx        ; <cx>=25/c-d - ст. часть
          add      bx,2         ; <bx>=25/c-d+2
          adc      cx,0
```

```
; мл. часть числителя
          mov       WORD PTR Numerator, BX
; ст. часть числителя
          mov       WORD PTR Numerator+2, CX
          ; готовимся к делению
          mov       ax,bx
          mov       dx,cx
; РЕЗУЛЬТАТ <ax>=(25/c-d+2)/(a*a-1+d)
          IDIV      Denominator
          mov       xI,ax
          ret
Lab3I   endp
          end
```

Как видите, код на Ассемблере действительно НЕ изменился. Теперь можно и протестировать нашу реализацию...

Тесты будут такие же, как и в варианте реализации 1.



ЗАМЕЧАНИЕ.

При стыковке модулей на Ассемблере с программами на алгоритмическом языке C++ с использованием среды программирования Borland C++ 3.1 можно брать и компиляторы **tasm.exe** версий выше, чем 3.1. Наверное, исследуя листинги в прилагаемых к данной книге материалах, вы уже обратили на это внимание.

Вариант 3. Особенности реализации в Borland C++ 5.02 и Turbo Assembler 4.1.

В этом варианте реализации программы на C++ мы в полной мере воспользуемся рассуждениями, приведенными при решении примера 6.2b. Добавим также и поддержку кириллицы (см. п. 6.1.2.8, пример 6.2rus). В отличие от предыдущих реализаций проверка на допустимый диапазон знаменателя и результата сделана НЕ в программе на C++, а в модуле на Ассемблере (применены команды сравнения и команды условного (**JNE** — переход, если НЕ равно) и безусловного перехода (**JMP**) — см. п.9.1.1 и 9.2). Модуль на Ассемблере выполнил студент второго курса ХАИ Устименко Вадим (автором настоящей книги добавлены ТОЛЬКО комментарии). Вы, уважаемый читатель, можете сравнить все эти реализации и сделать собственные выводы (а лучше собственную реализацию!).

Исходный текст модуля asmfile.asm

```
; Выполнил студент В. Устименко
; (гр. 627 ХАИ, 24.03.2001)
; Компиляция: tasm asmfile.asm /l /ml
; Вычислить заданное целочисленное выражение для
; исходных данных в знаковых форматах
; длиной 8 и 16 бит: signed char и int:
;        Вариант № 62) (25/c – d + 2)/(d + a*a-1);

.model large, C
CODESEG
```

```
Extrn c ass:byte
Extrn c cbs:byte
Extrn c dbs:byte

Extrn c aws:word
Extrn c cws:word
Extrn c dws:word

Extrn c n:word
Extrn c dn:word
Extrn c n4:Dword
Extrn c res_16:word
Extrn c f:byte

Public c proc_byte_s
Public c proc_word_s

proc_byte_s proc far
    mov ax, 25
    mov bl, cbs
    idiv bl   ; 25/c
    cbw
    mov bx,ax
    mov al,dbs
    cbw
    sub bx,ax      ;   25/c - d
    mov ax, 2
    add ax, bx     ;   25/c - d + 2
    mov n, ax      ; числитель
;==============================
    mov al, ass
    mov bl, ass
    imul bl        ; a*a
    mov bx,ax
    mov al,dbs
    cbw
    add ax, bx         ; a*a + d
    sub ax, 1          ; a*a + d - 1
    mov dn, ax         ; знаменатель
; Результат
    mov ax, n
    cwd
    mov bx, dn
    idiv bx
    mov res_16, ax
    ret
proc_byte_s endp
;====================================
proc_word_s proc far
    mov f,0
    mov ax, 25
```

```
    cwd
    mov bx, cws
    idiv bx         ; 25/c
    cwd
    mov word ptr n4,ax
    mov word ptr n4+2,dx
    mov ax,dws
    cwd
    sub word ptr n4, ax      ;  25/c – d (мл.часть)
    sbb word ptr n4+2,dx     ;  25/c – d (ст.часть)
    mov ax, 2
    cwd
    add word ptr n4, ax      ;  25/c – d + 2 (мл.часть)
    adc word ptr n4+2,dx     ;  25/c – d + 2 (ст.часть)
;------------------------
    mov ax, aws
    mov bx, aws
    imul bx         ; a*a
    mov cx,dx
    cwd
; Проверка на допустимый диапазон произведения a*a
    cmp cx,dx       ; проверка содержимого dx
    jne er4
    mov dn,ax

    mov ax,dws
    cwd
    add ax,dn
    adc dx,0        ; a*a + d

    mov cx,dx
    cwd
; Проверка на допустимый диапазон произведения a*a+d
    cmp cx,dx       ; проверка содержимого dx
    jne er4
    sub ax, 1       ; a*a + d -1
    mov dn, ax      ; знаменатель
    mov ax,word ptr n4
    mov dx,word ptr n4+2
    mov bx, dn
    idiv bx
    mov res_16, ax
    jmp SHORT en4
er4:mov f,1
en4:ret
proc_word_s endp
    end
```

Исходный текст программы Lab3c.cpp

```
/* Lab3c.cpp              Borland C++ 5.02
          Проект !!!!!! Application.exe === > DOS (standard) !!!!!!
                       asm+cpp файлы
          (c) Copyright 1998-2001 by Голубь Н.Г.
                   Лабораторная работа № 3.
  Вычислить заданное целочисленное выражение для исходных данных
     в знаковых форматах длиной 8 и 16 бит: signed char и int:
        Вариант № 62) (25/c — d + 2)/(d + a*a-1);
          Корректный ввод числовых символов.
             Поддержка КИРИЛЛИЦЫ.
*/

#include <iostream.h>
#include <conio.h>
#include <typeinfo.h>
#include "inputNum.h" // Ввод ЛЮБЫХ ЦЕЛЫХ чисел
#include "convert.h"  // Перекодировка КИРИЛЛИЦЫ Win—>DOS

typedef unsigned char byte;
const char* InputA = TXT(" Введите значение a: ");
const char* InputC = TXT(" Введите значение c: ");
const char* InputD = TXT(" Введите значение d: ");
const char* ResultC =   TXT("\n\n Результат в C++ = ");
const char* ResultASM = TXT("\n\n Результат в ASM = ");
const char* CONTINUE = TXT("\n\n Нажмите любую клавишу для продолжения...");
const char* CALC = TXT("\n\t  Вычислить вариант # 62) (25/c — d + 2)/(d + a*a-1);\n");
const char* SELECT = TXT("\n Выберите тип аргументов для вычисления:\n\n");
const char* CHAR =   TXT("\t 1 — Char (1 байт со знаком)\n");
const char* INT =    TXT("\t 2 — Int  (2 байта со знаком\n");
const char* EXIT =   TXT("\t 3 — Выход из этой программы...\n");
const char* RESULT = TXT("\n!!!!!!! Результат ");
const char* Numerator   = TXT("\n Числитель   ");
const char* Denominator = TXT("\n Знаменатель ");
const char* RANGE = TXT(" вне диапазона ");
const char* DIVIDE = TXT("\nОшибка!!!!! Деление на ноль!!!\n");
const char* Overfloat =
      TXT("\n!!!! Ошибка ПЕРЕПОЛНЕНИЯ при вычислениях в Ассемблере!!!\n");
const char* ARGUMENT = TXT("\n Тип аргументов ");

// ВНЕШНИЕ ассемблерные функции
extern "C"
{
 void proc_byte_s(void);
 void proc_word_s(void);
}

void title(void)
{
 cout << CALC;
}
```

```
void title(long int a, long int c, long int d)
{
// (25/c — d + 2)/(d + a*a-1)
 cout << "\n          (25/"<< c << " — " << d << " + 2)"
      << "\n ———————————————————————————————— "
      << "\n      (" << d << " + " << a << "*" << a << " — 1)\n";
}

void PressAnyKey()
{ cout <<  CONTINUE;
  getch();
}

char menu(void)
{
 cout << SELECT
        << CHAR
        << INT
        << EXIT;
 return getch();
}

//——————————— ГЛОБАЛЬНЫЕ переменные — передаются в Ассемблер
int  dn, res_16, n;
long n4;
char ass, cbs, dbs;
int  aws, cws, dws;

byte f;
//———————————————————————————————————————————

// ПАРАМЕТРИЗОВАННАЯ функция вычисления арифметического выражения
template <class Type>
int calculate(Type a, Type c, Type d)
{
   long res;
   long dn = d + (long)a * a — 1;
   if (!c||!dn)
      {
        cout << DIVIDE;
        return 0; // Деление на ноль!!!
      }
   else
    {
          long n = 25/c — (long)d + 2;
          res = n/dn;
          cout << ResultC << res;
          cout << Numerator << n;
          cout << Denominator << dn;
        }
  return 1;  // Ok!
}
```

```
void ProcChar()
{
 cout << ARGUMENT << " CHAR (-128..127) \n\n";
 while (input (InputA,ass));
 while (input (InputC,cbs));
 while (input (InputD,dbs));
 title(ass,cbs,dbs);
 if (calculate(ass,cbs,dbs))
  { proc_byte_s();
     if (!f) // Флаг ПЕРЕПОЛНЕНИЯ
     {
              cout << ResultASM << res_16;
              cout << Numerator << n;
         cout << Denominator << dn;
            }
          else cout << Overfloat;
  }
}

void ProcInt()
{
 cout << ARGUMENT << " INT (-32768..32767)\n\n";
 while (input (InputA,aws));
 while (input (InputC,cws));
 while (input (InputD,dws));
 title(aws,cws,dws);
 if (calculate(aws,cws,dws))
  { proc_word_s();
     if (!f)
     {
              cout << ResultASM << res_16;
              cout << Numerator << n4;
         cout << Denominator << dn;
            }
          else cout << Overfloat;
  }
}

int main(void)
{
 int t=0;
 char point;
 do{
     clrscr();
     title();
     point = menu();
     switch(point)
      {
        case '1':ProcChar();break;
        case '2':ProcInt(); break;
        case '3': return 0;
      }
     cout << "\n===================== test #" << ++t << " ==============\n";
     PressAnyKey();
   }while (point != '3');
}
```

Тестовые примеры:

```
            Вычислить вариант # 62) (25/c — d + 2)/(d + a*a-1);
Выберите тип аргументов для вычисления:
        1 — Char (1 байт со знаком)
        2 — Int  (2 байта со знаком)
        3 — Выход из этой программы...

Тип аргументов  CHAR (-128..127)
Введите значение a: -128
Введите значение c: 1
Введите значение d: -128
        (25/1 - -128 + 2)
      ———————————————————
      (-128 + -128*-128 - 1)

Результат в C++ = 0
Числитель   155
Знаменатель 16255

Результат в ASM = 0
Числитель   155
Знаменатель 16255
====================== test #1 ========================
Нажмите любую клавишу для продолжения...

            Вычислить вариант # 62) (25/c — d + 2)/(d + a*a-1);
Выберите тип аргументов для вычисления:
        1 — Char (1 байт со знаком)
        2 — Int  (2 байта со знаком)
        3 — Выход из этой программы...
Тип аргументов  CHAR (-128..127)
Введите значение a: 1
Введите значение c: 1
Введите значение d: 1
        (25/1 — 1 + 2)
        ——————————————
        (1 + 1*1 — 1)

Результат в C++ = 26
Числитель   26
Знаменатель 1
Результат в ASM = 26
Числитель   26

Знаменатель 1
====================== test #2 ========================
Нажмите любую клавишу для продолжения...

            Вычислить вариант # 62) (25/c — d + 2)/(d + a*a-1);
Выберите тип аргументов для вычисления:
        1 — Char (1 байт со знаком)
        2 — Int  (2 байта со знаком)
        3 — Выход из этой программы...
```

```
  Тип аргументов  CHAR (-128..127)
  Введите значение a: 1
  Введите значение c: -1
  Введите значение d: -1
             (25/-1 — -1 + 2)
            ─────────────────
              (-1 + 1*1 - 1)

  Результат в C++ = 22
  Числитель    -22
  Знаменатель -1
  Результат в ASM = 22
  Числитель    -22
  Знаменатель -1
==================== test #3 =====================
  Нажмите любую клавишу для продолжения...

            Вычислить вариант # 62) (25/c — d + 2)/(d + a*a-1);
  Выберите тип аргументов для вычисления:
          1 — Char (1 байт со знаком)
          2 — Int  (2 байта со знаком)
          3 — Выход из этой программы...

  Тип аргументов  INT (-32768..32767)
  Введите значение a: -2222
  Введите значение c: -1
  Введите значение d: 22222
             (25/-1 — 22222 + 2)
        ──────────────────────────
          (22222 + -2222*-2222 — 1)

  Результат в C++ = 0
  Числитель    -22245
  Знаменатель 4959505
!!!! Ошибка ПЕРЕПОЛНЕНИЯ при вычислениях в Ассемблере!!!
==================== test #4 =====================
  Нажмите любую клавишу для продолжения...

            Вычислить вариант # 62) (25/c — d + 2)/(d + a*a-1);
  Выберите тип аргументов для вычисления:
          1 — Char (1 байт со знаком)
          2 — Int  (2 байта со знаком)
          3 — Выход из этой программы...

Тип аргументов  INT (-32768..32767)
Введите значение a: qwe
      Ошибка! НЕ корректен формат числа!!!
Введите значение a: 2222222
      Ошибка! НЕ корректен диапазон для типа int
Введите значение a: 1
Введите значение c: -1
Введите значение d: 32767
          (25/-1 — 32767 + 2)
        ────────────────────
          (32767 + 1*1 — 1)
Результат в C++ = -1
Числитель    -32790
Знаменатель 32767
!!! Ошибка ПЕРЕПОЛНЕНИЯ при вычислениях в Ассемблере!!!
```

```
==================== test #5 =======================
Нажмите любую клавишу для продолжения...

             Вычислить вариант # 62) (25/c - d + 2)/(d + a*a-1);
 Выберите тип аргументов для вычисления:
          1 - Char (1 байт со знаком)
          2 - Int  (2 байта со знаком)
          3 - Выход из этой программы...

 Тип аргументов  INT (-32768..32767)
 Введите значение a: 111
 Введите значение c: -25
 Введите значение d: -32768
             (25/-25 - -32768 + 2)
             ------------------------
             (-32768 + 111*111 - 1)

 Результат в C++ = -1
 Числитель   32769
 Знаменатель -20448

 Результат в ASM = -1
 Числитель   32769
 Знаменатель -20448
===================== test #6 =======================
 Нажмите любую клавишу для продолжения...
```



ОБРАТИТЕ ВНИМАНИЕ

на достаточно безобидный **test #4**. Первый и второй вариант реализации с ним справился успешно, а здесь Ассемблер считать "НЕ захотел"...

Попробуйте сами протестировать этот вариант реализации и сделайте свои СОБСТВЕННЫЕ выводы.



Лабораторная работа № 4

Организация условных переходов (процессор i8086/i286)

Ставь себе лишь достижимые цели.

Квинт Гораций Флакк
(65-8 гг. до н.э.)

Цель работы

Вычислить заданное **условное целочисленное** выражение для данных в форматах **INTEGER (int)** и **WORD (unsigned int)**, используя команды сравнения, условного и безусловного переходов. Результат **X** — тоже **целочисленный и его диапазон (формат) зависит от специфики решаемого условного выражения**. Исходные данные должны вводиться корректно (с проверкой на область допустимых значений). Результат также должен быть проверен на область допустимых значений. Данные должны передаваться в подпрограммы (функции) как параметры.

Порядок работы

1) внимательно изучить свой вариант условного арифметического выражения;
2) написать на базовом алгоритмическом языке программу ввода исходных данных (с контролем допустимого диапазона), вычисления условного арифметического выражения и вывода полученного результата;
3) написать модуль вычисления условного арифметического выражения на языке Ассемблера;
4) встроить вызов этого модуля в программу на базовом алгоритмическом языке;
5) произвести тестовые проверки, отметить нормальные и аномальные результаты, сделать анализ результатов.

Варианты

1) $X = \begin{cases} (a-b)/a+1, & \text{если} \quad a > b, \\ 25, & \text{если} \quad a = b, \\ (a-5)/b, & \text{если} \quad a < b; \end{cases}$

2) $X = \begin{cases} (a-b)/a-3, & \text{если} \quad a > b, \\ 2, & \text{если} \quad a = b, \\ (a^3+1)/b, & \text{если} \quad a < b; \end{cases}$

3) $X = \begin{cases} b/a+5, & \text{если} \quad a < b, \\ -5, & \text{если} \quad a = b, \\ (a*a-b)/b, & \text{если} \quad a > b; \end{cases}$

4) $X = \begin{cases} a/b+10, & \text{если} \quad a < b, \\ -51, & \text{если} \quad a = b, \\ (a*b-4)/a, & \text{если} \quad a > b; \end{cases}$

5) $X = \begin{cases} a/b-1, & \text{если} \quad a > b, \\ -25, & \text{если} \quad a = b, \\ (b^3-5)/a, & \text{если} \quad a < b; \end{cases}$

6) $X = \begin{cases} a/b-1, & \text{если} \quad a > b, \\ -25, & \text{если} \quad a = b, \\ (b^3-5)/a, & \text{если} \quad a < b; \end{cases}$

7) $X = \begin{cases} 52*b/a+b, & \text{если} \quad a > b, \\ -125, & \text{если} \quad a = b, \\ (a-5)/b, & \text{если} \quad a < b; \end{cases}$

8) $X = \begin{cases} (a*b-1)/a, & \text{если} \quad a > b, \\ 255, & \text{если} \quad a = b, \\ (a-5)/b, & \text{если} \quad a < b; \end{cases}$

9) $X = \begin{cases} 1-b/a, & \text{если} \quad a > b, \\ -10, & \text{если} \quad a = b, \\ (a-5)/b, & \text{если} \quad a < b; \end{cases}$

10) $X = \begin{cases} a/b+31, & \text{если} \quad a > b, \\ -25, & \text{если} \quad a = b, \\ (5*b-1)/a, & \text{если} \quad a < b; \end{cases}$

11) $X = \begin{cases} (2+b)/a & \text{если} \quad a > b, \\ -2, & \text{если} \quad a = b, \\ (a-5)/b, & \text{если} \quad a < b; \end{cases}$

12) $X = \begin{cases} b/a+1, & \text{если} \quad a < b, \\ 25, & \text{если} \quad a = b, \\ (a^3-5)/b, & \text{если} \quad a > b; \end{cases}$

13) $X = \begin{cases} b/a+61, & \text{если} \quad a > b, \\ -5, & \text{если} \quad a = b, \\ (b-a)/b, & \text{если} \quad a < b; \end{cases}$

14) $X = \begin{cases} a/b+1, & \text{если} \quad a > b, \\ -2, & \text{если} \quad a = b, \\ (a-b)/a, & \text{если} \quad a < b; \end{cases}$

15) $X = \begin{cases} (3*a-5)/b, & \text{если} \quad a < b, \\ -4, & \text{если} \quad a = b, \\ (a^3+b)/a, & \text{если} \quad a > b; \end{cases}$

16) $X = \begin{cases} b/a-1, & \text{если} \quad a < b, \\ -295, & \text{если} \quad a = b, \\ (a-235)/b, & \text{если} \quad a > b; \end{cases}$

17) $X = \begin{cases} 2*a/b+1, & \text{если} \quad a > b, \\ -445, & \text{если} \quad a = b, \\ (b+5)/a, & \text{если} \quad a < b; \end{cases}$

18) $X = \begin{cases} a/b+1, & \text{если} \quad a > b, \\ a+25, & \text{если} \quad a = b, \\ (a*b-2)/a, & \text{если} \quad a < b; \end{cases}$

19) $X = \begin{cases} b/a+10, & \text{если} \quad a > b, \\ 3425, & \text{если} \quad a = b, \\ (2*a-5)/b, & \text{если} \quad a < b; \end{cases}$

20) $X = \begin{cases} (a*a-b)/a, & \text{если} \quad a > b, \\ -a, & \text{если} \quad a = b, \\ (a*b-1)/b, & \text{если} \quad a < b; \end{cases}$

21) $X = \begin{cases} (b+1)/a, & \text{если} \quad a > b, \\ -b, & \text{если} \quad a = b, \\ (a-5)/b, & \text{если} \quad a < b; \end{cases}$

22) $X = \begin{cases} a/b-1, & \text{если} \quad a < b, \\ 25-a, & \text{если} \quad a = b, \\ (b-5)/a, & \text{если} \quad a > b; \end{cases}$

23) $X = \begin{cases} b/a+2, & \text{если} \quad a > b, \\ -11, & \text{если} \quad a = b, \\ (a-8)/b, & \text{если} \quad a < b; \end{cases}$

24) $X = \begin{cases} a/b+2, & \text{если} \quad a > b, \\ 8, & \text{если} \quad a = b, \\ (b-9)/a, & \text{если} \quad a < b; \end{cases}$

25) $X = \begin{cases} (b+5)/a, & \text{если} \quad a < b, \\ -5, & \text{если} \quad a = b, \\ (b-a)/b, & \text{если} \quad a > b; \end{cases}$

26) $X = \begin{cases} a/b+1, & \text{если} \quad a < b, \\ -71, & \text{если} \quad a = b, \\ (a-b)/a, & \text{если} \quad a > b; \end{cases}$

27) $X = \begin{cases} b/a-7, & \text{если} \quad a > b, \\ 43, & \text{если} \quad a = b, \\ (a^3-b)/b, & \text{если} \quad a < b; \end{cases}$

28) $X = \begin{cases} -5+b/a, & \text{если} \quad a > b, \\ 45, & \text{если} \quad a = b, \\ (3*a-6)/b, & \text{если} \quad a < b; \end{cases}$

29) $X = \begin{cases} a/b+7, & \text{если} \quad a > b, \\ -125, & \text{если} \quad a = b, \\ (3*b+9)/a, & \text{если} \quad a < b; \end{cases}$

30) $X = \begin{cases} a/b-4, & \text{если} \quad a < b, \\ -55, & \text{если} \quad a = b, \\ (b-5)/a, & \text{если} \quad a > b; \end{cases}$

31) $X = \begin{cases} a/b+20, & \text{если} \quad a > b, \\ 110, & \text{если} \quad a = b, \\ (a-b)/a, & \text{если} \quad a < b; \end{cases}$

32) $X = \begin{cases} a/b+11, & \text{если} \quad a > b, \\ -11, & \text{если} \quad a = b, \\ (3*b-9)/a, & \text{если} \quad a < b; \end{cases}$

33) $X = \begin{cases} b/a-5, & \text{если } a > b, \\ 22, & \text{если } a = b, \\ (a-9)/b, & \text{если } a < b; \end{cases}$

34) $X = \begin{cases} 2*a/b+1, & \text{если } a > b, \\ -5, & \text{если } a = b, \\ (a^3-9)/a, & \text{если } a < b; \end{cases}$

35) $X = \begin{cases} b/a+2, & \text{если } a > b, \\ -57, & \text{если } a = b, \\ (a-b)/b, & \text{если } a < b; \end{cases}$

36) $X = \begin{cases} b/a+1, & \text{если } a > b, \\ -20, & \text{если } a = b, \\ (a-45)/b, & \text{если } a < b; \end{cases}$

37) $X = \begin{cases} b/a+1, & \text{если } a > b, \\ -44, & \text{если } a = b, \\ (a^3+b)/b, & \text{если } a < b; \end{cases}$

38) $X = \begin{cases} a/b+2, & \text{если } a > b, \\ -100, & \text{если } a = b, \\ (b-100)/a, & \text{если } a < b; \end{cases}$

39) $X = \begin{cases} a/b+201, & \text{если } a > b, \\ -800, & \text{если } a = b, \\ (a-5)/a, & \text{если } a < b; \end{cases}$

40) $X = \begin{cases} a/b+1, & \text{если } a > b, \\ -100, & \text{если } a = b, \\ (a*b-9)/a, & \text{если } a < b; \end{cases}$

41) $X = \begin{cases} a/b+1, & \text{если } a > b, \\ -425, & \text{если } a = b, \\ (3*a-b)/a, & \text{если } a < b; \end{cases}$

42) $X = \begin{cases} a/b-a, & \text{если } a < b, \\ -b, & \text{если } a = b, \\ (a*b-8)/a, & \text{если } a > b; \end{cases}$

43) $X = \begin{cases} a/b+5, & \text{если } a < b, \\ -b, & \text{если } a = b, \\ (b-9)/a, & \text{если } a > b; \end{cases}$

44) $X = \begin{cases} b^2/a+1, & \text{если } a > b, \\ 2*a, & \text{если } a = b, \\ (a-10)/b, & \text{если } a < b; \end{cases}$

45) $X = \begin{cases} b/a+8, & \text{если } a > b, \\ 455, & \text{если } a = b, \\ (a-b)/b, & \text{если } a < b; \end{cases}$

46) $X = \begin{cases} b/a+1, & \text{если } a > b, \\ -271, & \text{если } a = b, \\ (a-b)/b, & \text{если } a < b; \end{cases}$

47) $X = \begin{cases} a/b-37, & \text{если } a > b, \\ 3, & \text{если } a = b, \\ (a^3-b)/a, & \text{если } a < b; \end{cases}$

48) $X = \begin{cases} b/a-5, & \text{если } a > b, \\ -195, & \text{если } a = b, \\ (a-6)/b, & \text{если } a < b; \end{cases}$

49) $X=\begin{cases} a/b-7, & \text{если} \quad a>b, \\ -195, & \text{если} \quad a=b, \\ (b^3+9)/a, & \text{если} \quad a<b; \end{cases}$

50) $X=\begin{cases} a/b-4, & \text{если} \quad a<b, \\ -155, & \text{если} \quad a=b, \\ (b^3-5)/a, & \text{если} \quad a>b; \end{cases}$

51) $X=\begin{cases} a/b-42, & \text{если} \quad a<b, \\ -11, & \text{если} \quad a=b, \\ (a^3-8)/a, & \text{если} \quad a>b; \end{cases}$

52) $X=\begin{cases} b/a-141, & \text{если} \quad a<b, \\ -131, & \text{если} \quad a=b, \\ (3*a-9)/b, & \text{если} \quad a>b; \end{cases}$

53) $X=\begin{cases} 4/a+5, & \text{если} \quad a>b, \\ 28, & \text{если} \quad a=b, \\ (a^3-9)/(b-2), & \text{если} \quad a<b; \end{cases}$

54) $X=\begin{cases} a/4-b, & \text{если} \quad a>b, \\ -57, & \text{если} \quad a=b, \\ (a^3-5)/b, & \text{если} \quad a<b; \end{cases}$

55) $X=\begin{cases} a/b-2, & \text{если} \quad a>b, \\ -7, & \text{если} \quad a=b, \\ (a^3-3)/a, & \text{если} \quad a<b; \end{cases}$

56) $X=\begin{cases} a/b+1, & \text{если} \quad a>b, \\ -25, & \text{если} \quad a=b, \\ (b-45)/a, & \text{если} \quad a<b; \end{cases}$

57) $X=\begin{cases} a/b-1, & \text{если} \quad a>b, \\ -414, & \text{если} \quad a=b, \\ (a^3-b)/a, & \text{если} \quad a<b; \end{cases}$

58) $X=\begin{cases} b/a-2, & \text{если} \quad a>b, \\ -140, & \text{если} \quad a=b, \\ (a-100)/b, & \text{если} \quad a<b; \end{cases}$

59) $X=\begin{cases} a/b+21, & \text{если} \quad a>b, \\ -82, & \text{если} \quad a=b, \\ (a-b)/a, & \text{если} \quad a<b; \end{cases}$

60) $X=\begin{cases} b/a-1, & \text{если} \quad a>b, \\ -120, & \text{если} \quad a=b, \\ (a*b-9)/b, & \text{если} \quad a<b. \end{cases}$

Контрольные вопросы

1. Особенности выполнения изучаемых базовых команд процессора i8086/i286.
2. Команды безусловного перехода.
3. Команды условного перехода. Организация разветвлений.
4. Разница в организации условных переходов для знаковых и беззнаковых данных.
5. Соглашения о передаче параметров, принятые для используемого в лабораторной работе алгоритмического языка.
6. Состояние стека при выполнении процедур и функций с параметрами.
7. Ассемблирование и дизассемблирование команд на своих примерах.
8. Понятие о байтах кода операции, способах адресации, регистрах и смещениях.

9. Необходимость в контроле диапазона целочисленных данных при вводе.
10. Диапазон допустимых значений для целочисленных переменных.

Пример решения типового варианта

Вариант № 62.

Вычислить заданное **условное целочисленное** выражение:

$$X = \begin{cases} b/a-4, & \text{если} \quad a < b, \\ 25, & \text{если} \quad a = b, \\ (a^3-5)/b, & \text{если} \quad a > b; \end{cases}$$

для данных в форматах **INTEGER (int)** и **WORD (unsigned int)**, используя команды сравнения, условного и безусловного переходов. Результат **X** — тоже **целочисленный и его диапазон (формат) зависит от специфики решаемого условного выражения**. Исходные данные должны вводиться корректно (с проверкой на область допустимых значений). Результат также должен быть проверен на область допустимых значений. Данные должны передаваться в подпрограммы (функции) как параметры.

Решение

Сделаем для нашего примера главную программу на алгоритмическом языке С++ (Borland C++ 5.02). Здесь тоже, как и в предыдущей лабораторной работе, будут продемонстрированы ПРИНЦИПЫ решения ДАННОЙ задачи. ОПТИМИЗАЦИЮ ПРОГРАММЫ, разобравшись с основами, вы сможете сделать и сами.

При выполнении данной лабораторной работы нам понадобятся в основном материалы главы 9 и, конечно, глав 5-6. Кроме того, здесь будет полезен и опыт, приобретенный нами при выполнении лабораторной работы № 3.

Проиллюстрируем наши действия по шагам.

ШАГ 0.
Анализ особенностей задачи

1. У нашей задачи возможны две ситуации деления на ноль:
   - **b/a-4, если a<b.** Например, **a=0, b=1.**
   - **(a^3-5)/b, если a>b.** Например, **a=1, b=0.**

   В то же время ситуация, когда **a=b=0**, является совершенно нормальной.
2. При **a>b** возможно переполнение при вычислении **a*a*a,** поэтому в С++ результирующую переменную **x** нужно сделать, как минимум, хотя бы типа **long**, чтобы это переполнение зафиксировать и НЕ допустить вычислений в Ассемблере (иначе возможна ситуация "ДЕЛЕНИЕ НА НОЛЬ" — см. п. 5.2.4).
3. В нашей задаче имеет смысл сделать **входные и выходные данные одного типа.** У вашего варианта, вполне вероятно, будут другие, свои, СПЕЦИФИЧЕСКИЕ особенности, отличные от тех, которые мы рассматриваем. **Учитесь эти особенности ВИДЕТЬ!** А для этого — решайте больше примеров, набирайтесь опыта...

ШАГ 1.
Главная программа на С++

Решим сначала нашу задачу на С++. Функции на Ассемблере предусмотрим, но пока подключать их НЕ будем (**их вызовы закомментируем** — это общепринятый у программистов способ ставить "заглушки") — тем более, что их пока и нет.

Исходный текст программы Lab4.cpp — вариант 1.

```
/* Lab4.cpp              Borland C++ 5.02
        Проект !!!!!! Application.exe === > DOS (standard) !!!!!!
                    (c) Copyright 1998-2001 by Голубь Н.Г.
                     Лабораторная работа # 4.
   Вычислить заданное условное целочисленное выражение:
        b/a-4           , a<b
     y= 25              , a=b
        (a^3-5)/b       , a>b
для данных в форматах INTEGER (int) и WORD (unsigned int), используя команды сравне-
ния, условного и безусловного переходов. Результат X — тоже целочисленный и его диа-
пазон (формат) в данном конкретном случае совпадает с форматом исходных данных.
Исходные данные должны вводиться корректно (с проверкой на область допустимых зна-
чений). Результат также должен быть проверен на область допустимых значений. Данные
должны передаваться в подпрограммы (функции) как параметры.
   */

   #include <limits.h>
   #include <iostream.h>
   #include <conio.h>
   #include <fstream.h>

   /*============================================================*/
   //(c) Copyright 1998 by Артем Астафьев (629 группа ХАИ)
   #define   test(a) ((a>>15)+1)&~1 // Контроль -32768<=a<=32767, тип int
   #define   testU(a) a>>16        // Контроль     0<=a<=65535, тип unsigned int
   /*============================================================*/

   const char* CONTINUE = "\n\n Press any key to continue...";
   const char* SELECT = "\n Choose type of arguments:\n\n";
   const char* UNINT =  "\t 1 — unsigned int\n";
   const char* INT =    "\t 2 — int\n";
   const char* EXIT =   "\t 3 — Exit from this program...\n";

   // Превышение диапазона для INT при вычислении ВЫРАЖЕНИЯ
    char* ERROR_RANGE_EXPRESSION="\nC++: !!! ERROR RANGE EXPRESSION !!!\n";

   // ГЛОБАЛЬНАЯ переменная — для Ассемблера
    unsigned char f;

   //!!!!!!!!!!! Описание внешних ASM-процедур !!!!!!!!!!!
   extern "C"
   {  int lab4I (int a, int b);
      unsigned int lab4U (unsigned int a, unsigned int b);
   }
   //!!!!!!!!!!!!!!!!!!!!!!!!!!

   void title(int& t)
```

```
{
    clrscr();
    cout << "\n================== test #" << i++ << " ==================\n";
    cout <<"\n            Variant 62:     "<< endl;
    cout << "            b/a-4       , a<b "<< endl;
    cout << "     y=   25          , a=b "<< endl;
    cout << "           (a^3-5)/b , a>b "<< endl;
}
void PressAnyKey()
{ cout <<  CONTINUE;
  getch();
}
char menu(void)
{
 cout << SELECT
        << UNINT
        << INT
        << EXIT;
 return getch();
}
// ввод ЛЮБОГО ТИПА переменной — через шаблон
template <class TypeVariable>
int input (TypeVariable& k, char* var) // КОРРЕКТНЫЙ ВВОД
{  // объявление и привязка ЛОКАЛЬНЫХ вх./вых. потоков
    ifstream my_inp ("CON");
    ofstream my_out ("CON");
    ostream* old_tie = my_inp.tie(&my_out);  // привязка потоков
    my_out << "\nEnter value " << var << " ——>  ";
    my_out.flush();   // принудит. освобождение буфера
    my_inp >> k;
    // Анализ битов ввода
          switch (my_inp.rdstate())
                    { case ios::goodbit:  return 0;
                      case ios::eofbit :  return 0;
                      case ios::failbit:
                      case ios::badbit :
                              my_out << "\n!!!!! Error !!!!!";
                              my_out << "\n continue ....." << endl;
                              return 1;
                    }
}

/*========= ПАРАМЕТРИЗОВАННАЯ функция на С++ ==================*/
template <class TypeVariable>
int func(TypeVariable a, TypeVariable b, long CONST_MIN=SHRT_MIN,
                                          long CONST_MAX=SHRT_MAX)
 {
   long x;
    if (a<b)
    {
       cout << "            b/a-4     ; " << a << " <  " << b << endl;
       if (a) x=b/a-4;
        else
       {
```

```
           cout << "C++: ERROR!!! Divide by ZERO!!!";
           return 1;
          }
       }
      else
        if (a>b)
         {
          cout << "          (a^3-5)/b ; " << a << " >  " << b << endl;
          if(b) x=(a*a*a-5)/b;
           else
               {
                cout << "C++: ERROR!!! Divide by ZERO!!!";
                return 1;
               }
              }
         else
         { cout << a << " =  " << b << endl;
           x=25;
          }
// Проверка на ДОПУСТИМЫЙ диапазон результата
      if (x > CONST_MIN && x < CONST_MAX)
       {
         cout<<"CPP:   Result =  " << (TypeVariable)x << endl;
         return 1;  // Все нормально!
       }
      else
       {  cout<<ERROR_RANGE_EXPRESSION;
          cout<<"  Result =  " << x << endl;
          return 0;
        }
  }
/*=============================================================*/

void ProcUint(long a, long b, unsigned yU)
{
//  Корректный ввод целого числа и контроль его диапазона [0..65535]
      do {  while(input(a,"a (unsigned int)"));
           } while (testU(a));
      do {  while(input(b,"b (unsigned int)"));
           } while (testU(b));
    // Вызов C++ функции
    if (func((unsigned int)a,(unsigned int)b,0L,(long)UINT_MAX))
     {
/*        yU=lab4U((unsigned int)a,(unsigned int)b);  // Вызов процедуры на ассемблере
          if (!f) cout <<"\nASM:   Result =  "<< yU << endl;
          else cout << "\nASM: ERROR!!! Divide by ZERO!!!"; */
     }
}

void ProcInt(long a, long b, int y)
{
    //  Корректный ввод целого числа и контроль его диапазона [-32768..32767]
    do {  while(input(a,"a (int)"));
        } while (test(a));
    do {  while(input(b,"b (int)"));
       } while (test(b));
```

```
    // Вызов C++ функции
    if (func((int)a,(int)b))
     {
/*         y=lab4l((int)a,(int)b);  // Вызов процедуры на ассемблере
          if (!f) cout <<"\nASM:  Result =  " << y << endl;
          else cout << "\nASM: ERROR!!! Divide by ZERO!!!";   */
     }
}

int main()
{ long int a,b;
   int y,t=1;
   unsigned yU;
   char point;
   for(;;)
   {
      title(t);
      point = menu();
      switch(point)
      {
          case "1":ProcUint(a,b,yU);break;
          case "2":ProcInt(a,b,y); break;
          case "3": return 0;
      }
      PressAnyKey();
   }
}
```

ШАГ 2.

Тестирование программы — вариант 1

```
===================== test #1 ======================
             Variant 62:
            b/a-4      , a<b
       y=   25         , a=b
            (a^3-5)/b  , a>b

 Choose type of arguments:
          1 – unsigned int
          2 – int
          3 – Exit from this program...

Enter value a (unsigned int) —>  1
Enter value b (unsigned int) —>  40000
             b/a-4     ; 1 <  40000
CPP:   Result =   39996

 Press any key to continue...

===================== test #2 ======================
             Variant 62:
            b/a-4      , a<b
       y=   25         , a=b
            (a^3-5)/b  , a>b
 Choose type of arguments:
```

```
          1 - unsigned int
          2 - int
          3 - Exit from this program...

 Enter value a (int) —>  0
 Enter value b (int) —>  0
 0 =  0
 CPP:   Result =  25

  Press any key to continue...

 ===================== test #3 =======================
              Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
              (a^3-5)/b  , a>b

  Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

 Enter value a (unsigned int) —>  111
 Enter value b (unsigned int) —>  1
              (a^3-5)/b ; 111 >  1
 CPP:   Result =  56906  ?????????

  Press any key to continue...

 ===================== test #4 =======================
              Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
              (a^3-5)/b  , a>b

  Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

 Enter value a (int) —>  111
 Enter value b (int) —>  1
              (a^3-5)/b ; 111 >  1
 CPP:   Result =  -8630  ?????????
  Press any key to continue...

 ===================== test #5 =======================
              Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
              (a^3-5)/b  , a>b

  Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

 Enter value a (unsigned int) —>  0
 Enter value b (unsigned int) —>  1
              b/a-4     ; 0 <  1
 C++: ERROR!!! Divide by ZERO!!!
```

```
  Press any key to continue...

===================== test #6 =======================
            Variant 62:
            b/a-4      , a<b
      y=    25         , a=b
            (a^3-5)/b  , a>b

  Choose type of arguments:
          1 - unsigned int
          2 - int
          3 - Exit from this program...

Enter value a (int) —>  1
Enter value b (int) —>  0
            (a^3-5)/b ; 1 >  0
C++: ERROR!!! Divide by ZERO!!!

  Press any key to continue...

===================== test #7 =======================
            Variant 62:
            b/a-4      , a<b
      y=    25         , a=b
            (a^3-5)/b  , a>b

  Choose type of arguments:
          1 - unsigned int
          2 - int
          3 - Exit from this program...

Enter value a (int) —>  -1
Enter value b (int) —>  32767
             b/a-4     ; -1 <  32767
CPP:    Result =   32765 ??????????

  Press any key to continue...

===================== test #8 =======================
            Variant 62:
            b/a-4      , a<b
      y=    25         , a=b
            (a^3-5)/b  , a>b

  Choose type of arguments:
          1 - unsigned int
          2 - int
          3 - Exit from this program...

Enter value a (unsigned int) —>  22222
Enter value b (unsigned int) —>  55555
             b/a-4     ; 22222 <  55555
CPP:    Result =   65534 ???????????

  Press any key to continue...
```

Тесты test #3, test #4, test #7 и test #8 дали НЕВЕРНЫЙ результат:

1) НЕ было зафиксировано ПЕРЕПОЛНЕНИЕ при умножении **a*a*a** (test #3, test #4).

2) НЕ было зафиксировано ПЕРЕПОЛНЕНИЕ ДИАПАЗОНА для **int** при вычислении **b/a-4 = 32767/(-1)-4 = -32771** (test #7).

3) НЕ было зафиксировано ПЕРЕПОЛНЕНИЕ ДИАПАЗОНА для **unsigned int** при вычислении **b/a-4 = 55555/22222-4 = 2-4=-2** (test #8).

ШАГ 3.
Исправление ошибок в программе — переход к варианту 2

Все перечисленные ошибки связаны с особенностями вычислений, когда при арифметических операциях используются младшие типы данных (в нашем случае **int** и **unsigned int**). А результат присваивается старшему типу (в нашем случае **long int**). Об этом шла речь с самого начала — см. п. 5.2.3 (пояснение к примеру 5.6). Этот момент обобщен в приложении 8 (теперь вы можете понять все, что в нем написано, не правда ли?).

Таким образом, нужно исправить два оператора (добавить **явное** преобразование типов данных):

```
if (a) x=(long)b/a-4;
if (b) x=((long)a*a*a-5)/b;
```

Получим вариант 2 — см. в прилагаемых к книге материалах.

ШАГ 4.
Тестирование программы — вариант 2

```
===================== test #1 ======================
              Variant 62:
              b/a-4      , a<b
       y=     25         , a=b
             (a^3-5)/b   , a>b

 Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

Enter value a (unsigned int) —>  111
Enter value b (unsigned int) —>  1
             (a^3-5)/b ; 111 >  1
C++: !!! ERROR RANGE EXPRESSION !!!
  Result =  1367626

 Press any key to continue...
===================== test #2 ======================
              Variant 62:
              b/a-4      , a<b
       y=     25         , a=b
             (a^3-5)/b   , a>b

 Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

Enter value a (int) —>  111
Enter value b (int) —>  1
             (a^3-5)/b ; 111 >  1

C++: !!! ERROR RANGE EXPRESSION !!!
  Result =  1367626
```

```
 Press any key to continue...
====================== test #3 ========================
            Variant 62:
            b/a-4      , a<b
      y=    25         , a=b
           (a^3-5)/b   , a>b

 Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

Enter value a (int) —>  -1
Enter value b (int) —>  32767
             b/a-4     ; -1 <  32767

C++: !!! ERROR RANGE EXPRESSION !!!
  Result =  -32771

 Press any key to continue...
====================== test #4 ========================
            Variant 62:
            b/a-4      , a<b
      y=    25         , a=b
           (a^3-5)/b   , a>b

 Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

Enter value a (unsigned int) —>  22222
Enter value b (unsigned int) —>  55555
             b/a-4     ; 22222 <  55555

C++: !!! ERROR RANGE EXPRESSION !!!
  Result =  -2

 Press any key to continue...
====================== test #5 ========================
            Variant 62:
            b/a-4      , a<b
      y=    25         , a=b
           (a^3-5)/b   , a>b

 Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

Enter value a (int) —>  22222
Enter value b (int) —>  -22222
           (a^3-5)/b ; 22222 >  -22222
CPP:   Result =  1519 ???????????????

 Press any key to continue...
```

Предыдущие тесты, на которых были ошибки, теперь выполняются нормально, но дал ошибку **test #5**. Эта ошибка "говорит" о том, что для таких, достаточно больших значений переменной **a** зафиксировать переполнение можно только, если тип результата будет превышать тип **long**. Поэтому принимаем тип результата, например, **double**.

ШАГ 5.
Исправление ошибок в программе — переход к варианту 3

Опять исправляем нашу программу:

```
double x;
if(b) x=((double)a*a*a-5)/b;
```

Получаем вариант 3 — см. в прилагаемых к книге материалах.

ШАГ 6.
Тестирование программы — вариант 3

```
===================== test #1 =======================
             Variant 62:
             b/a-4     , a<b
       y=    25        , a=b
             (a^3-5)/b , a>b

 Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

Enter value a (int) —>  22222
Enter value b (int) —>  -22222
            (a^3-5)/b ; 22222 >  -22222

C++: !!! ERROR RANGE EXPRESSION !!!
  Result =  -4.93817e+08

 Press any key to continue...

===================== test #2 =======================
             Variant 62:
             b/a-4     , a<b
       y=    25        , a=b
             (a^3-5)/b , a>b

 Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

Enter value a (int) —>  111
Enter value b (int) —>  -30000
            (a^3-5)/b ; 111 >  -30000
CPP:   Result =  -45

 Press any key to continue...
===================== test #3 =======================
             Variant 62:
             b/a-4     , a<b
       y=    25        , a=b
             (a^3-5)/b , a>b
```

```
  Choose type of arguments:
            1 - unsigned int
            2 - int
            3 - Exit from this program...
Enter value a (int) —>  1000
Enter value b (int) —>  -30000
            (a^3-5)/b ; 1000 >  -30000
C++: !!! ERROR RANGE EXPRESSION !!!
   Result =  -33333.3

  Press any key to continue...
```

Ну, что же, теперь вроде все благополучно. Можно заняться и Ассемблером.

ШАГ 7. ASM-процедура вычисления условного арифметического выражения для 16-битных знаковых данных — lab4I

```
title Lab_4
; CopyRight by Голубь Н.Г., 1998, 2001
.model large,C
LOCALS @@
;                    b/a-4       , a<b
;          y=       25           , a=b
;                   (a^3-5)/b   , a>b
CODESEG
EXTRN   C f:BYTE        ; Флажок "Деление на ноль"
public C lab4I
lab4I proc far          ; Данные знаковые!!!
ARG a:WORD,b:WORD RETURNS y:WORD
  ;================ Начало процедуры =========
  mov f,0               ;ok!
  mov ax,a              ;<ax>=a
  mov bx,b              ;<bx>=b
  cmp ax,bx             ; сравнение a~b
  jg @@2                ;a>b goto @@2
  jl @@1                ;a<b goto @@1
;================== a = b ==================
  mov ax,25
  jmp short @@Exit
;================== a < b ==================
@@1:
  cmp ax,0              ; сравнение a~0
  JE   @@ERROR          ; a=0 goto @@ERROR
  mov ax,bx             ; <ax>=b
  CWD
  IDIV a                ; <ax>=<dx:ax>/a
  SUB  ax,4             ; <ax>=b/a-4
  jmp short @@Exit
;===================a > b ==================
@@2:
  cmp bx,0              ; сравнение b~0
  JE   @@ERROR          ; a=0 goto @@ERROR
  Imul a                ; <dx:ax>=a*a
```

```
   Imul a             ; <dx:ax>=a*a*a
   add  ax,-5         ; <ax>=<ax>-5    (мл.часть)
   adc  dx,0FFFFh ; <dx>=<dx> -(ст.часть числа -5)
   Idiv bx            ; <ax>=(a*a*a-5)/b
   jmp  short @@Exit
;=============== Деление на ноль =========
@@ERROR:
     mov f,1
     ret
;=============== Получение результата =========
@@Exit:
     mov y,ax
     ret
  ;=========== Конец процедуры ===============
lab4I endp
;
public C lab4U
lab4U proc far   ; Данные БЕЗзнаковые!!!
ARG a:WORD,b:WORD RETURNS y:WORD
  ;================ Начало процедуры =========
     ret
  ;=========== Конец процедуры ===============
lab4U endp
end
```

ШАГ 8.

Тестирование и анализ программы с ASM-модулем lab4I

Откроем в программе на C++ соответствующие комментарии и протестируем полученный вариант программы.

```
===================== test #1 ======================
            Variant 62:
            b/a-4      , a<b
      y=    25         , a=b
            (a^3-5)/b  , a>b

 Choose type of arguments:
          1 - unsigned int
          2 - int
          3 - Exit from this program...

Enter value a (int) --> 111
Enter value b (int) --> -300000
Enter value b (int) --> -30000
           (a^3-5)/b ; 111 >  -30000
CPP:    Result =  -45
ASM:    Result =  -45

 Press any key to continue...

===================== test #2 ======================
            Variant 62:
```

```
              b/a-4 , a<b
        y=    25          , a=b
              (a^3-5)/b  , a>b

  Choose type of arguments:
              1 - unsigned int
              2 - int
              3 - Exit from this program...

 Enter value a (int) —>  0
 Enter value b (int) —>  0
 0 =  0
 CPP:    Result =  25
 ASM:    Result =  25

  Press any key to continue...
 ===================== test #4 =======================
              Variant 62:
              b/a-4 , a<b
        y=    25          , a=b
              (a^3-5)/b  , a>b

  Choose type of arguments:
              1 - unsigned int
              2 - int
              3 - Exit from this program...

 Enter value a (int) —>  1
 Enter value b (int) —>  32767
              b/a-4     ; 1 <  32767
 CPP:    Result =  32763
 ASM:    Result =  32763

  Press any key to continue...
 ===================== test #5 =======================
              Variant 62:
              b/a-4 , a<b
        y=    25          , a=b
              (a^3-5)/b  , a>b

  Choose type of arguments:
              1 - unsigned int
              2 - int
              3 - Exit from this program...

 Enter value a (int) —>  100
 Enter value b (int) —>  1
              (a^3-5)/b ; 100 >  1
 C++: !!! ERROR RANGE EXPRESSION !!!
    Result =  999995
  Press any key to continue...
 ===================== test #6 =======================
              Variant 62:
              b/a-4      , a<b
        y=    25          , a=b
              (a^3-5)/b  , a>b
  Choose type of arguments:
              1 - unsigned int
```

```
              2 - int
              3 - Exit from this program...

Enter value a (int) —>  200
Enter value b (int) —>  -30000
              (a^3-5)/b ; 200 >  -30000

CPP:    Result =  -266
ASM:    Result =  170

 Press any key to continue...
```

Опять ошибка. Кто "врет"? Вычисляем вручную: 200*200*200= 8000000. Далее 8000000/(-30000) = **-266. C++ считает верно.** Результат ПЕРВОГО умножения 200*200 = 40000 превысил диапазон для 16-разрядного числа. Затем опять следует команда умножения, а, как известно (см. табл. 5.9), в нашем случае множимое должно находиться в регистре **AX** (т.е. содержимое регистра **DX** при последующем умножении мы учитывать пока НЕ умеем!). Поэтому для **знаковых данных** нужно проверять в Ассемблере флаг **OF** (см. табл. 9.4) и выдавать в программе на C++ соответствующее сообщение.

ШАГ 9.

Исправление ASM-процедуры вычисления условного арифметического выражения для 16-битных знаковых данных — lab4l

На ветви **a>b** при вычислении **(a*a*a-5)/b** команд умножений у нас две. Проверять флаг **OF** мы будем после первой команды умножения. После второго умножения мы нормально учитываем регистр **DX**. Для индикации в C++ этой ситуации придется добавить в Ассемблере еще одно значение флажка: **f=2** ("Переполнение при 16-разрядном умножении") и его анализ в C++:

```
if (!f) cout <<"\nASM:   Result =  "<< yU << endl;
  else
     if (f==1) cout << "\nASM: ERROR!!! Divide by ZERO!!!";
          else cout << "\nASM: Sorry!!! Overflow at 16-digit multiplication!!!";
```

Код на Ассемблере в исправленном и дополненном виде будет иметь следующий вид:

```
;==================a > b =================
@@2:
  cmp bx,0          ; сравнение b~0
  JE   @@ERROR      ; a=0 goto @@ERROR
  Imul a            ; <dx:ax>=a*a
  Jo    @@ERRORm    ; Переполнение при умножении
     Imul a         ; <dx:ax>=a*a*a
  add   ax,-5       ; <ax>=<ax>+(-5)     (мл.часть)
  adc   dx,0FFFFh   ; <dx>=<dx>+(ст.часть числа -5)
  Idiv bx           ; <ax>=(a*a*a-5)/b
  jmp short @@Exit
;=============== Деление на ноль =========
@@ERROR:
    mov f,1
    ret
;========= Переполнение при 16-разрядном умножении
  @@ERRORm:
```

```
mov f,2
ret
```

ШАГ 10.
Тестирование исправленной программы с ASM-модулем lab4I

```
======================== test #2 ===================
              Variant 62:
             b/a-4       , a<b
       y=    25          , a=b
             (a^3-5)/b , a>b

  Choose type of arguments:
             1 - unsigned int
             2 - int
             3 - Exit from this program...

Enter value a (int) —>  100
Enter value b (int) —>  -11111
             (a^3-5)/b ; 100 >  -11111

CPP:    Result =  -90
ASM:    Result =  -90

  Press any key to continue...
===================== test #3 ======================
              Variant 62:
             b/a-4       , a<b
       y=    25          , a=b
             (a^3-5)/b , a>b

  Choose type of arguments:
             1 - unsigned int
             2 - int
             3 - Exit from this program...

Enter value a (int) —>  130
Enter value b (int) —>  -22222
             (a^3-5)/b ; 130 >  -22222
CPP:    Result =  -98
ASM:    Result =  -98

  Press any key to continue...
===================== test #4 ======================
              Variant 62:
             b/a-4       , a<b
       y=    25          , a=b
             (a^3-5)/b   , a>b

  Choose type of arguments:
             1 - unsigned int
             2 - int
             3 - Exit from this program...

Enter value a (int) —>  150
Enter value b (int) —>  -30000
             (a^3-5)/b ; 150 >  -30000
```

```
CPP:    Result =  -112
ASM:    Result =  -112
 Press any key to continue...
===================== test #5 =======================
            Variant 62:
           b/a-4      , a<b
      y=   25         , a=b
           (a^3-5)/b  , a>b
 Choose type of arguments:
          1 - unsigned int
          2 - int
          3 - Exit from this program...
Enter value a (int) —>  200
Enter value b (int) —>  -20000
          (a^3-5)/b ; 200 >  -20000
CPP:    Result =  -399
ASM: Sorry!!! Overflow at 16-digit multiplication!!!
 Press any key to continue...
===================== test #6 =======================
            Variant 62:
           b/a-4      , a<b
      y=   25         , a=b
           (a^3-5)/b  , a>b
 Choose type of arguments:
          1 - unsigned int
          2 - int
          3 - Exit from this program...
Enter value a (int) —>  160
Enter value b (int) —>  -22222
          (a^3-5)/b ; 160 >  -22222
CPP:    Result =  -184
ASM:    Result =  -184
 Press any key to continue...
===================== test #7 =======================
            Variant 62:
           b/a-4      , a<b
      y=   25         , a=b
           (a^3-5)/b  , a>b
 Choose type of arguments:
          1 - unsigned int
          2 - int
          3 - Exit from this program...
Enter value a (int) —>  170
Enter value b (int) —>  -111
          (a^3-5)/b ; 170 >  -111
C++: !!! ERROR RANGE EXPRESSION !!!
  Result =  -44261.2
 Press any key to continue...

===================== test #8 =======================

            Variant 62:
           b/a-4      , a<b
      y=   25         , a=b
           (a^3-5)/b  , a>b
 Choose type of arguments:
          1 - unsigned int
```

```
            2 – int
            3 – Exit from this program...

Enter value a (int) —>  170
Enter value b (int) —>  -22222
             (a^3-5)/b ; 170 >  -22222
CPP:    Result =  -221
ASM:    Result =  -221

 Press any key to continue...

====================== test #12 ======================
              Variant 62:
             b/a-4      , a<b
        y=   25         , a=b
             (a^3-5)/b  , a>b

 Choose type of arguments:
             1 – unsigned int
             2 – int
             3 – Exit from this program...

Enter value a (int) —>  250
Enter value b (int) —>  -30000
             (a^3-5)/b ; 250 >  -30000
CPP:    Result =  -520
ASM: Sorry!!! Overflow at 16-digit multiplication!!!

 Press any key to continue...
```

Сейчас результаты на наших тестах получились нормальные.

Теперь можно заняться реализацией на Ассемблере процедуры обработки БЕЗЗНАКОВЫХ 16-РАЗРЯДНЫХ данных. Здесь будет, в общем, **ПОЛНАЯ аналогия в рассуждениях**, что и со знаковыми данными, поэтому эти шаги мы приводить НЕ будем. В приведенном ниже листинге исходного ASM-модуля **жирным шрифтом выделены команды, отличающиеся для знаковых и беззнаковых данных.** Как видите, отличающихся команд не так уж и много. Мы много времени затратили на отладку знакового варианта, набрались опыта, набили шишек. Поэтому беззнаковый вариант должен восприниматься уже легче. Файл тестов для этого варианта **VarUnInt.txt** можно посмотреть в прилагаемых к книге материалах.

Реализация в Borland C++ 5.02 и Turbo Assembler 4.1 — окончательный вариант

Исходный текст программы Lab4.cpp

```
/* Lab4.cpp                 Borland C++ 5.02
        Проект !!!!!! Application.exe === > DOS (standard) !!!!!!
                       (c) Copyright 1998-2001 by Голубь Н.Г.
                        Лабораторная работа # 4.
Вычислить заданное условное целочисленное выражение:
              b/a-4          , a<b
         y=   25             , a=b
            (a^3-5)/b        , a>b
для данных в форматах INTEGER (int) и WORD (unsigned int), используя команды сравнения, условного и безусловного переходов. Результат X — тоже целочисленный и его диапазон (формат) в данном конкретном случае совпадает с форматом исходных данных.
Исходные данные должны вводиться корректно (с проверкой на область допустимых значений). Результат также должен быть проверен на область допустимых значений. Данные должны передаваться в подпрограммы (функции) как параметры.
```

```
*/

#include <limits.h>
#include <iostream.h>
#include <conio.h>
#include <fstream.h>

/*===========================================================*/
//(c) Copyright 1998 by Артем Астафьев (629 группа ХАИ)
#define   test(a) ((a>>15)+1)&~1 // Контроль -32768<=a<=32767, тип int
#define   testU(a) a>>16        // Контроль       0<=a<=65535, тип unsigned int
/*===========================================================*/

const char* CONTINUE = "\n\n Press any key to continue...";
const char* SELECT = "\n Choose type of arguments:\n\n";
const char* UNINT =  "\t 1 — unsigned int\n";
const char* INT =    "\t 2 — int\n";
const char* EXIT =   "\t 3 — Exit from this program...\n";

// Превышение диапазона для INT при вычислении ВЫРАЖЕНИЯ
 char* ERROR_RANGE_EXPRESSION="\nC++: !!! ERROR RANGE EXPRESSION !!!\n";

// ГЛОБАЛЬНАЯ переменная — для Ассемблера
 unsigned char f;

//!!!!!!!!!!! Описание внешних ASM-процедур !!!!!!!!!!!
extern "C"
{   int lab4I (int a, int b);
     unsigned int lab4U (unsigned int a, unsigned int b);
}
//!!!!!!!!!!!!!!!!!!!!!!!!!!

void title(int& t)
{
   clrscr();
   cout << "\n================== test #" << t++ << " ================\n";
    cout <<"\n           Variant 62:     "<< endl;
    cout << "            b/a-4       , a<b "<< endl;
    cout << "      y=   25          , a=b "<< endl;
    cout << "           (a^3-5)/b , a>b "<< endl;
}

void PressAnyKey()
{ cout <<  CONTINUE;
  getch();
}

char menu(void)
{
 cout << SELECT
        << UNINT
        << INT
        << EXIT;
 return getch();
}

// ввод ЛЮБОГО ТИПА переменной — через шаблон
```

```
template <class TypeVariable>
int input (TypeVariable& k, char* var) // КОРРЕКТНЫЙ ВВОД
{ // объявление и привязка ЛОКАЛЬНЫХ вх./вых. потоков
    ifstream my_inp ("CON");
    ofstream my_out ("CON");
    ostream* old_tie = my_inp.tie(&my_out);  // привязка потоков
            my_out << "\nEnter value " << var << " ——>  ";
            my_out.flush();   // принудит. освобождение буфера
            my_inp >> k;
           // Анализ битов ввода
       switch (my_inp.rdstate())
                  { case ios::goodbit:  return 0;
                    case ios::eofbit :  return 0;
                    case ios::failbit:
                    case ios::badbit :
                            my_out << "\n!!!!! Error !!!!!";
                            my_out << "\n continue ....." << endl;
                            return 1;
                  }
}

/*================== ПАРАМЕТРИЗОВАННАЯ функция на C++ ======*/
template <class TypeVariable>
int func(TypeVariable a, TypeVariable b,        long CONST_MIN=SHRT_MIN,
                                                long CONST_MAX=SHRT_MAX)
 {
          double x;
     if (a<b)
     {
      cout << "            b/a-4    ; " << a << " <  " << b << endl;
      if (a) x=(long)b/a-4; //!!!!!!!!!!!
        else
         {
          cout << "C++: ERROR!!! Divide by ZERO!!!";
          return 1;
         }
      }
      else
        if (a>b)
         {
          cout << "          (a^3-5)/b ; " << a << " >  " << b << endl;
          if(b) x=((double)a*a*a-5.)/b; //!!!!!!!!!!
           else
               {
                 cout << "C++: ERROR!!! Divide by ZERO!!!";
                 return 1;
               }
            }
         else
         { cout << a << " =  " << b << endl;
           x=25;
          }
// Проверка на ДОПУСТИМЫЙ диапазон результата
     if (x > CONST_MIN && x < CONST_MAX)
       {
         cout<<"CPP:   Result =  " << (TypeVariable)x << endl;
```

```
        return 1;  // Все нормально!
       }
      else
       {  cout<<ERROR_RANGE_EXPRESSION;
          cout<<"  Result =  " << x << endl;
          return 0;
        }
  }
/*==================================================================*/
void ProcUint(long a, long b, unsigned yU)
{
//  Корректный ввод целого числа и контроль его диапазона [0..65535]
     do {  while(input(a,"a (unsigned int)"));
        } while (testU(a));
     do {  while(input(b,"b (unsigned int)"));
        } while (testU(b));
    // Вызов C++ функции
    if (func((unsigned int)a,(unsigned int)b,0L,(long)UINT_MAX))
     {
         yU=lab4U((unsigned int)a,(unsigned int)b);  // Вызов процедуры на ассемблере
         if (!f) cout <<"\nASM:   Result =  "<< yU << endl;
          else
                   if (f==1) cout << "\nASM: ERROR!!! Divide by ZERO!!!";
                     else cout << "\nASM: Sorry!!! Overflow at 16-digit multiplication!!!";
     }
}

void ProcInt(long a, long b, int y)
{
     //  Корректный ввод целого числа и контроль его диапазона [-32768..32767]
     do {  while(input(a,"a (int)"));
        } while (test(a));
     do {  while(input(b,"b (int)"));
        } while (test(b));
    // Вызов C++ функции
    if (func((int)a,(int)b))
     {
         y=lab4I((int)a,(int)b);  // Вызов процедуры на ассемблере
         if (!f) cout <<"\nASM:   Result =  " << y << endl;
         else
          if (f==1) cout << "\nASM: ERROR!!! Divide by ZERO!!!";
               else cout << "\nASM: Sorry!!! Overflow at 16-digit multiplication!!!";
     }
}

int main()
{  long int a,b;
   int y,t=1;
   unsigned yU;
   char point;
   for(;;)
    {
      title(t);
      point = menu();
      switch(point)
```

```
    {
      case "1":ProcUint(a,b,yU);break;
      case "2":ProcInt(a,b,y); break;
      case "3": return 0;
     }
    PressAnyKey();
  }
}
```

Исходный текст модуля Lab_4.asm

```
title Lab_4 (CopyRight by Голубь Н.Г., 1998, 2001)
.model large,C
LOCALS @@
;                 b/a-4      , a<b
;           y=    25         , a=b
;                 (a^3-5)/b , a>b
CODESEG
EXTRN  C f:BYTE  ; Флажок "Деление на ноль": f=1
; Флажок "Переполнение при 16-разрядном умножении" f=2
public C lab4I
lab4I proc far  ; Данные знаковые!!!
ARG a:WORD,b:WORD RETURNS y:WORD
  ;================ Начало процедуры =========
  mov f,0        ; ok!
  mov ax,a       ; <ax>=a
  mov bx,b       ; <bx>=b
  cmp ax,bx      ; сравнение a~b
  jg @@2         ; a>b goto @@2
  jl @@1         ; a<b goto @@1
;================== a = b ==================
  mov ax,25
  jmp short @@Exit
;================== a < b ==================
@@1:
  cmp ax,0       ; сравнение a~0
  JE  @@ERROR    ; a=0 goto @@ERROR
  mov ax,bx      ; <ax>=b
  CWD
  IDIV a         ; <ax>=<dx:ax>/a
  SUB  ax,4      ; <ax>=b/a-4
  jmp short @@Exit
;==================a > b ==================
@@2:
  cmp bx,0       ; сравнение b~0
  JE  @@ERROR    ; a=0 goto @@ERROR
  Imul a         ; <dx:ax>=a*a
  Jo  @@ERRORm   ; Переполнение при знаковом умножении
  Imul a         ; <dx:ax>=a*a*a
  add  ax,-5     ; <ax>=<ax>+(-5)    (мл.часть)
  adc  dx,0FFFFh     ; <dx>=<dx>+(ст.часть числа -5)
  Idiv bx            ; <ax>=(a*a*a-5)/b
```

```
   jmp short @@Exit
;============== Деление на ноль =========
@@ERROR:
      mov f,1
      ret
;======= Переполнение при 16-разрядном умножении
@@ERRORm:
      mov f,2
      ret
;=============== Получение результата =========
@@Exit:
      mov y,ax
      ret
;=========== Конец процедуры ===============
lab4I endp
;
public C lab4U
lab4U proc far  ; Данные БЕЗзнаковые!!!
ARG a:WORD,b:WORD RETURNS y:WORD
   ;================ Начало процедуры =========
   mov f,0       ; ok!
   mov ax,a      ; <ax>=a
   mov bx,b      ; <bx>=b
   cmp ax,bx     ; сравнение a~b
   jA @@2        ; a>b goto @@2
   jB @@1        ; a<b goto @@1
;================= a = b =================
   mov ax,25
   jmp short @@Exit
;================= a < b =================
@@1:
   cmp ax,0      ; сравнение a~0
   JE  @@ERROR   ; a=0 goto @@ERROR
   mov ax,bx     ; <ax>=b
   XOR dx,dx
   DIV a    ; <ax>=<dx:ax>/a - ДЕЛЕНИЕ БЕЗЗНАКОВОЕ!!!!
   SUB  ax,4     ; <ax>=b/a-4
   jmp short @@Exit
;==================a > b =================
@@2:
   cmp bx,0      ; сравнение b~0
   JE  @@ERROR   ; a=0 goto @@ERROR
   mul a         ; <dx:ax>=a*a
;Переполнение при БЕЗзнаковом умножении
   Jo   @@ERRORm
   mul a              ; <dx:ax>=a*a*a
   add  ax,-5         ; <ax>=<ax>+(-5)    (мл.часть)
   adc  dx,0FFFFh     ; <dx>=<dx>+(ст.часть числа -5)
   div bx ; <ax>=(a*a*a-5)/b - ДЕЛЕНИЕ БЕЗЗНАКОВОЕ!!!!
   jmp short @@Exit
```

```
;=============== Деление на ноль =========
@@ERROR:
     mov f,1
     ret
;========== Переполнение при 16-разрядном умножении
@@ERRORm:
     mov f,2
     ret
;=============== Получение результата =========
@@Exit:
     mov y,ax
     ret
  ;=========== Конец процедуры ================
lab4U endp
end
```

Результаты окончательного тестирования:

```
==================== test #1 ======================
             Variant 62:
             b/a-4      , a<b
       y=    25         , a=b
            (a^3-5)/b   , a>b

 Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

Enter value a (unsigned int) —>  1
Enter value b (unsigned int) —>  65535
             b/a-4      ; 1 <  65535
CPP:   Result =  65531
ASM:   Result =  65531

 Press any key to continue...

==================== test #2 ======================
             Variant 62:
             b/a-4      , a<b
       y=    25         , a=b
            (a^3-5)/b   , a>b

 Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

Enter value a (int) —>  1
Enter value b (int) —>  -32768
            (a^3-5)/b ; 1 >  -32768
CPP:   Result =  0
ASM:   Result =  0

 Press any key to continue...

==================== test #3 ======================
             Variant 62:
```

```
              b/a-4      , a<b
        y=    25         , a=b
             (a^3-5)/b   , a>b

  Choose type of arguments:
            1 - unsigned int
            2 - int
            3 - Exit from this program...

 Enter value a (int) —>  1
 Enter value b (int) —>  32767
              b/a-4      ; 1 <  32767
 CPP:    Result =  32763
 ASM:    Result =  32763

  Press any key to continue...

 ====================== test #4 ========================
              Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
             (a^3-5)/b   , a>b

  Choose type of arguments:
            1 - unsigned int
            2 - int
            3 - Exit from this program...

 Enter value a (unsigned int) —>  250
 Enter value b (unsigned int) —>  249
             (a^3-5)/b ; 250 >  249
 CPP:    Result =  62750
 ASM:    Result =  62750

  Press any key to continue...

 ====================== test #5 ========================
              Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
             (a^3-5)/b   , a>b

  Choose type of arguments:
            1 - unsigned int
            2 - int
            3 - Exit from this program...

 Enter value a (int) —>  250
 Enter value b (int) —>  249
             (a^3-5)/b ; 250 >  249
 C++: !!! ERROR RANGE EXPRESSION !!!
    Result =  62751

  Press any key to continue...

 ====================== test #6 ========================
              Variant 62:
```

```
             b/a-4       , a<b
        y=   25          , a=b
             (a^3-5)/b   , a>b

  Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

 Enter value a (int) —> 150
 Enter value b (int) —> 149
             (a^3-5)/b ; 150 > 149
 CPP:   Result = 22650
 ASM:   Result = 22650

  Press any key to continue...

 ===================== test #7 =======================
             Variant 62:
             b/a-4       , a<b
        y=   25          , a=b
             (a^3-5)/b   , a>b

  Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

 Enter value a (int) —> 150
 Enter value b (int) —> 1
             (a^3-5)/b ; 150 > 1
 C++: !!! ERROR RANGE EXPRESSION !!!
   Result = 3.375e+06

  Press any key to continue...

 ===================== test #8 =======================
             Variant 62:
             b/a-4       , a<b
        y=   25          , a=b
             (a^3-5)/b   , a>b

  Choose type of arguments:
           1 - unsigned int
           2 - int
           3 - Exit from this program...

 Enter value a (int) —> 180
 Enter value b (int) —> 179
             (a^3-5)/b ; 180 > 179
 CPP:   Result = 32580
 ASM:   Result = 32580

  Press any key to continue...

 ===================== test #9 =======================
             Variant 62:
```

```
             b/a-4      , a<b
        y=   25         , a=b
             (a^3-5)/b  , a>b

  Choose type of arguments:
             1 - unsigned int
             2 - int
             3 - Exit from this program...

 Enter value a (unsigned int) —>  0
 Enter value b (unsigned int) —>  0
 0 =  0
 CPP:   Result =  25
 ASM:   Result =  25

  Press any key to continue...

 ====================== test #10 ========================
             Variant 62:
             b/a-4      , a<b
        y=   25         , a=b
            (a^3-5)/b   , a>b

  Choose type of arguments:
             1 - unsigned int
             2 - int
             3 - Exit from this program...

 Enter value a (unsigned int) —>  0
 Enter value b (unsigned int) —>  44444
             b/a-4      ; 0 <  44444
 C++: ERROR!!! Divide by ZERO!!!
 ASM: ERROR!!! Divide by ZERO!!!

  Press any key to continue...

 ====================== test #11 ========================
             Variant 62:
             b/a-4      , a<b
        y=   25         , a=b
            (a^3-5)/b   , a>b

  Choose type of arguments:
             1 - unsigned int
             2 - int
             3 - Exit from this program...

 Enter value a (unsigned int) —>  2222
 Enter value b (unsigned int) —>  0
            (a^3-5)/b ; 2222 >  0
 C++: ERROR!!! Divide by ZERO!!!
 ASM: ERROR!!! Divide by ZERO!!!

  Press any key to continue...

 ====================== test #12 ========================
             Variant 62:
```

```
              b/a-4        , a<b
        y=    25           , a=b
             (a^3-5)/b    , a>b
   Choose type of arguments:

             1 - unsigned int
             2 - int
             3 - Exit from this program...

 Enter value a (int) —>  250
 Enter value b (int) —>  -30000
             (a^3-5)/b ; 250 >  -30000
 CPP:   Result =  -520
 ASM: Sorry!!! Overflow at 16-digit multiplication!!!

   Press any key to continue...

 ====================== test #13 ========================
              Variant 62:
              b/a-4        , a<b
        y=    25           , a=b
             (a^3-5)/b    , a>b

   Choose type of arguments:
             1 - unsigned int
             2 - int
             3 - Exit from this program...

 Enter value a (int) —>  300
 Enter value b (int) —>  -30000
             (a^3-5)/b ; 300 >  -30000
 CPP:   Result =  -899
 ASM: Sorry!!! Overflow at 16-digit multiplication!!!

   Press any key to continue...
```

Получилось относительно неплохо. На этом мы и остановимся. Попробуйте теперь протестировать вы, уважаемые мои читатели...

? ВОПРОС

Почему мы НЕ изменили оператор:

if (a) x=(**long**)b/a-4;

Что будет, если мы сделаем этот оператор таким:

if (a) x=(**double**)b/a-4;

Если вы знаете ответ на этот вопрос, — поздравляю! Если нет, ничего страшного, — все у вас еще впереди!



Лабораторная работа № 5

Организация циклов и работа с целочисленными одномерными массивами (процессор i8086/i286)

Не уклоняйся от дела, но и не суетись через меру.

Квинт Гораций Флакк (65-8 гг. до н.э.)

Цель работы

Задав одномерный массив целочисленных данных **A** в одном из заданных форматов (**unsigned char — BYTE, unsigned short int — WORD, char — SHORTINT, short int — INTEGER** *или* **long int — LONGINT**), *реализовать обработку* массива, *как указано в варианте. Длина* массива **N**. Исходные данные задать самостоятельно, учитывая формат элементов массива **A**.

В программе на С++ (ПАСКАЛЬ) должны быть предусмотрены функции ввода-вывода элементов массива А и его обработки. Исходные данные должны вводиться корректно и с проверкой на область допустимых значений. Тип результата определяется из контекста задачи.

Порядок работы

1) внимательно изучить свой вариант обработки элементов массива;
2) написать на базовом алгоритмическом языке программу ввода исходных данных (с контролем допустимого диапазона), обработки элементов массива и вывода полученного результата;
3) написать модуль обработки элементов массива на языке Ассемблера;
4) встроить вызов этого модуля в программу на базовом алгоритмическом языке;
5) произвести тестовые проверки, отметить нормальные и аномальные результаты, сделать анализ результатов.

Варианты

1 — unsigned char (BYTE); 2 — unsigned short int (WORD); 3 — char (SHORTINT); 4 — short int (INTEGER); 5 — long int (LONGINT):

Найти, сколько элементов массива **A={a[i]}** удовлетворяют условию: **c <= a[i] <= d**.

6 — unsigned char (BYTE); 7 — unsigned short int (WORD); 8 — char (SHORTINT); 9 — short int (INTEGER):

Найти произведение элементов массива **A={a[i]}**, удовлетворяющих условию: **c <= a[i] <= d**.

10 — long int (LONGINT); 11 — char (SHORTINT); 12 — short int (INTEGER):

Найти, сколько отрицательных элементов массива **A={a[i]}** удовлетворяют условию: **c <= a[i] <= d**.

13 — char (SHORTINT); 14 — short int (INTEGER):

Найти сумму кубов всех отрицательных элементов массива **A={a[i]}**, удовлетворяющих условию: **a[i] >= c*d.**

15 — long int (LONGINT); 16 — char (SHORTINT); 17 — short int (INTEGER):

Найти, сколько положительных элементов массива **A={a[i]}** удовлетворяют условию: **c <= a[i] <= d**.

18 — char (SHORTINT); 19 — short int (INTEGER):

Найти сумму квадратов всех положительных элементов массива **A={a[i]}**, удовлетворяющих условию: **a[i] >= d/c**.

20 — unsigned char (BYTE); 21 — unsigned short int (WORD):

Найти произведение квадратов первых **k** элементов массива **A={a[i]}**, удовлетворяющих условию: **a[i] >= c + d**.

22 — char (SHORTINT); 23 — short int (INTEGER); 24 — long int (LONGINT):

Найти, сколько положительных, отрицательных и нулевых элементов в массиве **A={a[i]}** удовлетворяют условию: **c <= a[i] <= d**.

25 — char (SHORTINT); 26 — short int (INTEGER):

Найти произведение квадратов отрицательных элементов массива **A={a[i]}**, удовлетворяющих условию: **a[i] >= c/d** .

27 — char (SHORTINT); 28 — short int (INTEGER):

Найти произведение последних **L** отрицательных элементов в массиве **A={a[i]}**, удовлетворяющих условию: **c <= a[i] <= d.**

29 — unsigned char (BYTE); 30 — unsigned short int (WORD); 31 — char (SHORTINT); 32 — short int (INTEGER); 33-long int (LONGINT):

Найти сумму первых **K** элементов массива **A={a[i]}**, удовлетворяющих условию: **c <= a[i] <= d**.

34 — unsigned char (BYTE); 35 — unsigned short int (WORD);
36 — char (SHORTINT); 37 — short int (INTEGER); 38-long int (LONGINT):
Найти количество элементов массива **A={a[i]}**, удовлетворяющих условию: **a[i] <= c/d.**

39 — unsigned char (BYTE); 40 — unsigned short int (WORD);
41 — char (SHORTINT); 42 — short int (INTEGER); 43-long int (LONGINT):
Найти сумму элементов массива **A={a[i]}**, удовлетворяющих условию:
c <= a[i] <= d.

44 — char (SHORTINT); 45 — short int (INTEGER):
Найти сумму последних **L** положительных элементов в массиве **A={a[i]}.**

46 — char (SHORTINT); 47 — short int (INTEGER):
Найти произведение положительных элементов массива **A={a[i]}**, удовлетворяющих условию: **c <= a[i] <= d.**

48 — char (SHORTINT); 49 — short int (INTEGER):
Найти произведение отрицательных элементов массива **A={a[i]}**, удовлетворяющих условию: **c <= a[i] <= d.**

50 — char (SHORTINT); 51 — short int (INTEGER):
Найти сумму кубов всех положительных элементов массива **A={a[i]}.**

52 — char (SHORTINT); 53 — short int (INTEGER):
Найти сумму квадратов всех отрицательных элементов массива **A={a[i]}.**

54 — char (SHORTINT); 55 — short int (INTEGER):
Найти произведение квадратов всех положительных элементов массива **A={a[i]}.**

56 — char (SHORTINT); 57 — short int (INTEGER):
Найти произведение квадратов всех отрицательных элементов массива **A={a[i]}.**

58 — char (SHORTINT); 59 — short int (INTEGER); 60 — long int (LONGINT):
Найти сумму первых **K** отрицательных элементов массива **A={a[i]}**, удовлетворяющих условию: **c <= a[i] <= d.**

Контрольные вопросы

1. Особенности выполнения изучаемых базовых команд процессора i8086/i286.
2. Команды безусловного перехода.
3. Команды условного перехода. Организация разветвлений.
4. Разница в организации условных переходов для знаковых и беззнаковых данных.
5. Команды управления циклом.
6. Соглашения о передаче параметров, принятые для используемого в лабораторной работе алгоритмического языка.
7. Состояние стека при выполнении процедур и функций с параметрами.

8. Ассемблирование и дизассемблирование команд на своих примерах.
9. Понятие о байтах кода операции, способах адресации, регистрах и смещениях.
10. Необходимость в контроле диапазона целочисленных данных при вводе.
11. Диапазон допустимых значений для целочисленных переменных:
char (SHORTINT), unsigned char (BYTE), short int (INTEGER), unsigned short int (WORD), int, unsigned int, unsigned long int (LONGINT).

Пример решения типового варианта

Вариант 62 — short int (INTEGER):

Найти произведение последних **L** положительных элементов в массиве **A={a[i]}**.

Длина массива **N**. Исходные данные задать самостоятельно, учитывая формат элементов массива **A**.

В программе на С++ (ПАСКАЛЬ) должны быть предусмотрены функции ввода-вывода элементов массива **A** и его обработки. Исходные данные должны вводиться корректно и с проверкой на область допустимых значений. Тип результата определяется из контекста задачи.

Реализация в Borland С++ 5.02 и Turbo Assembler 4.1

Решение

Сделаем для нашего примера главную программу на алгоритмическом языке С++ (Borland С++ 5.02) для 16-разрядной платформы MS DOS. В этом случае тип элементов массива **short int** совпадает с **int**. Здесь так же, как и в предыдущей лабораторной работе, будут продемонстрированы основные ПРИНЦИПЫ решения ДАННОЙ задачи. ОПТИМИЗАЦИЮ ПРОГРАММЫ, разобравшись с основами, вы сможете сделать и сами.

При выполнении данной лабораторной работы нам понадобятся в основном материалы главы 9 и глав 5-6. Кроме того, здесь будет полезен и опыт, приобретенный нами при выполнении лабораторных работ № 3 и №4. Мы уже набили достаточно шишек, теперь дело должно пойти быстрее.

Проиллюстрируем наши действия по шагам.

ШАГ 0.
Анализ особенностей задачи

1. Реализуем ручной ввод элементов массива, если их количество не превышает 10, в противном случае будем их генерировать. Примем **N=100**, поскольку нужно вычислить произведение. Если бы нужно было вычислить сумму, то значение **N** можно было бы взять и побольше, например, **N=1000.**
2. В нашей задаче довольно "заковыристый" алгоритм вычисления произведения элементов с КОНЦА массива. Мы с ним сначала разберемся на С++, а потом по образу и подобию сделаем в Ассемблере.
3. Чтобы легче было следить за элементами массива, сделаем визуализацию процесса получения произведения как в С++, так и в Ассемблере.

4. В нашей задаче (с точки зрения Ассемблера) имеет смысл сделать **результат (произведение положительных элементов массива)** типа **unsigned int**, чтобы немного раздвинуть диапазон результата. Сделать тип результата **long int** было бы лучше, но мы пока на Ассемблере не умеем перемножать данные **<long int>* <int>**. У вашего варианта, дорогой мой читатель, вполне вероятно, будут другие, свои, СПЕЦИФИЧЕСКИЕ особенности, отличные от тех, которые мы рассматриваем. **Учитесь эти особенности ВИДЕТЬ!** А для этого – решайте больше примеров, набирайтесь опыта... Другого пути нет!

ШАГ 1.
Главная программа на С++

Решим сначала нашу задачу на С++, взяв за основу пример 9.10. Функцию на Ассемблере предусмотрим, но пока подключать ее НЕ будем — ее вызов из С++ закомментируем.

Исходный текст программы Lab5.cpp

```
/* Lab5.cpp            Borland C++ 5.02
        Проект !!!!!! Application.exe === > DOS (standard) !!!!!!
                (c) Copyright 1998-2001 by Голубь Н.Г.
                        Лабораторная работа # 5.
      Найти произведение последних L положительных элементов массива
            int array[array_size], 2<=array_size<=100
*/
#include <iomanip.h> // манипулятор setw(8)
#include <conio.h> //getch(); clrscr();
#include <fstream.h>
#include <stdlib.h> // void randomize(void); int random (int)
#include "convert.h"  // Перекодировка КИРИЛЛИЦЫ Win—>DOS
const char* DATA_ERROR = TXT("Ошибка при вводе данных!!!\n");
const char* REPEAT = TXT("Повторите еще раз, пожалуйста...\n");
const char* VALUE = TXT("Значение должно быть от ");
const char* TO = TXT(" до ");
const char* VALUE_RANGE = TXT("Значение превысило допустимый диапазон\n");
const char* TITLE =
      TXT("Найти произведение последних L положительных элементов массива\n");
const char* ENTER_LEN = TXT("Введите длину массива или ");
const char* MEMORY = TXT("НЕВОЗМОЖНО распределить память для массива!!!\n");
const char* MULT = TXT("Введите количество умножаемых элементов\n");
const char* NO_ELEM = TXT("В данном массиве НЕТ положительных элементов!!!\n");
const char* RESULT_RANGE =
              TXT("Ошибка!!! Результат превышает допустимый диапазон: 65535\n");
const char* ASM_CALC =
            TXT("\nАссемблер: Произведение до конца вычислить НЕ удалось!!!\n");
const char* RES_ASM = TXT("Произведение БЕЗ последнего элемента: ");
const int N=100;
//ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
int array[N];
unsigned int array_size;
char exist=0;
extern "C"
{
```

```
    unsigned int far calc(int L);
          void RangeError(int i, int buf);
}
///////// Визуализация вычислений в C++ и на Ассемблере ///////////////
//
void RangeError(int j, int buf)
{
  if (j==1) cout << buf;
    else cout << "*" << buf;
}
///////////////////// КОРРЕКТНЫЙ ВВОД /////////////////////
//
template <class typData>
int input (typData& k)
{
      ifstream my_inp ("CON");
      my_inp >> k;

          switch (my_inp.rdstate())
                        { case ios::failbit:
                           case ios::badbit:
                                  cout << DATA_ERROR << REPEAT;
                                  cout.flush();
                                  return 1;
                          default:
                                  return 0; // ok!
                        }
}
//////////////// ВВОД С КОНТРОЛЕМ ДИАПАЗОНА ///////////////////
//
template <class typData>
int InputNum(typData& z, const long MIN=2L, const long MAX=N)
{
   long int temp;  //!!!!!!!!!!
   int exit_func=0;
   while (exit_func==0)
   {
      cout << VALUE << MIN << TO << MAX << " ===> ";
      cout.flush();
          while (input(temp));
         if ((temp>=MIN)&&(temp<=MAX)) exit_func = 1;
        else
        {
                 textcolor(LIGHTRED);
                 clreol();
                 cout << VALUE_RANGE;
                 textcolor(GREEN);
                 clreol();
                 cout.flush();
        }
   }
   z=(typData)temp;
   return exit_func;
}
////////////////////// ВВОД ЭЛЕМЕНТОВ МАССИВА /////////////
```

```
//
void inputArray(int A[], int array_size)
{
   randomize(); int j=1;
   for (int i=0; i<array_size; i++)
    if (array_size<11)
        {
                cout << "Enter A[" << i << "]"<< endl;
                InputNum(A[i],-32768L, 32767L);
         }
      else
      { j = -j;
        array[i]= j*random (30000);  //!!!!!!!!!!!!!!!!!!!!!!!!!!
        }
}
////////////////////// ВЫВОД ЭЛЕМЕНТОВ МАССИВА //////////////////
//
void outputArray(const int A[], int n)
{                                              cout                                        <<
"\n=================================================================\n"
                  << "         Array: \n";
  for (int i=0; i<n; i++)
        cout << setw(8) << A[i];
  cout << endl;
 }
////////////////////// ОСНОВНАЯ ЗАДАЧА /////////////////////////
//
long calc_c(const int array[], const int l, const int array_size)
{
   long res = 1;
   exist=0;
   int i=array_size-1;
   int j=0; // счетчик положительных элементов
   while (i>=0)
    {
      if (array[i]>0)
      {
        j++;
         RangeError(j, array[i]); // Визуализация вычислений
         res *= array[i];
         exist=1;
         if (j==l) return res;  // Найдено L элементов
      }
     i--;
   }
  return res;
}
//////////////////////////////////////////////////////////////
// main program
void main()
{
 int l,t=1;
 for(;;)
 {
   textcolor(WHITE);
```

```
    clrscr();
    cout << "\n========= test #" << t++ << "===============\n"
         << TITLE << "int array[array_size], 2<=array_size<=10000\n";
    cout << ENTER_LEN << "    Ctrl-C (exit)" << endl;
    // Ввод числа элементов массива
    InputNum(array_size);
    // Ввод элементов массива
    inputArray(array, array_size);
    outputArray(array, array_size);
    cout << MULT;
    cout.flush();
    // Ввод числа перемножаемых элементов массива
    InputNum(l, 1, array_size);
    textcolor(WHITE);
    // Вычисления на C++
    cout << "\nC++ function calculation\n";
    long res = calc_c(array, l, array_size);
    if (exist == 0) cout << "C++: " << NO_ELEM << endl;
      else
      {
            cout << " = " << res << endl;
            if (res > 65535L)cout << RESULT_RANGE;
      }
    // Вычисления на Ассемблере
/*  пока НЕ делаем!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    cout << "\nAsm function calculation\n";
    unsigned resAsm = calc(l);
    if (!exist) cout << " = " << resAsm << endl;
      else if (exist == -1) cout << "ASM: " << NO_ELEM << endl;
        else if (exist == 1)cout << ASM_CALC << endl;
*/
    cout.flush();
    getch();
  }
}
```

Шаг 2. Тестирование программы – вариант 1

```
========= test #1===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 20
=============================================================
            Array:
  -29261   16797  -17544     459  -16539    5647  -23147
  15308       -16487   13509
  -21486   22488  -16692   27225  -21140   18663  -28168
  17116       -27877   19336
Введите количество умножаемых элементов
Значение должно быть от 1 до 20 ===> 4
C++ function calculation
19336*17116*18663*27225 = -632156896 ??????????????
Нам повезло - мы сразу попали на ошибку!
```

1. Знак минус для произведения положительных элементов НИКАК НЕ МОЖЕТ БЫТЬ. Значит, нужно менять тип результата, чтобы справиться с анализом диапазона. Сделаем его тип **double**. Т.е. заменим в нашей программе три строки (**они выделены жирным шрифтом**).
2. Число 30000 для генератора случайных чисел тоже очень большое — надо думать и о том, как мы справимся с вычислениями в Ассемблере. Сделаем это число 300.

Шаг 3.
Исправление ошибок в программе – переход к варианту 2

```
array[i]= j*random (300);
double calc_c(const int array[], const int l, const int
   array_size)
{
    double res = 1;    //!!!!!!!!!!!!!!!!!
............
}
double res = calc_c(array, l, array_size);
```

```
Шаг 4. Тестирование программы - вариант 2
========= test #1===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 20
================================================================
             Array:
-295      137      -77      123      -143      200      -119
   150      -198      197
-199      122      -237      80      -265      11      -45
   21      -291      214
Введите количество умножаемых элементов
Значение должно быть от 1 до 20 ===> 11
C++ function calculation
214*21*11*80*122*197*150*200*123*137 = 4.80495e+19
Ошибка!!! Результат превышает допустимый диапазон: 65535
========= test #2===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 5
Enter A[0]
Значение должно быть от -32768 до 32767 ===> -2
Enter A[1]
Значение должно быть от -32768 до 32767 ===> 22
Enter A[2]
Значение должно быть от -32768 до 32767 ===> 3
Enter A[3]
Значение должно быть от -32768 до 32767 ===> 11
Enter A[4]
Значение должно быть от -32768 до 32767 ===> -6
================================================================
```

```
          Array:
      -2        22        3        11        -6
Введите количество умножаемых элементов
Значение должно быть от 1 до 5 ===> 2
C++ function calculation
11*3 = 33
```

Этот вариант немного лучше, но для генератора случайных чисел возьмем число 130 (оно немного больше предельного числа для типа **char**). Теперь можно переходить к Ассемблеру.

ШАГ 5. ASM-процедура calc

Исходный текст модуля Func.asm

```
      title Lab_5
; CopyRight by Голубь Н.Г., 1998, 2001
    .MODEL LARGE, C
    LOCALS @@
    .DATA
; ГЛОБАЛЬНЫЕ переменные
    EXTRN C array:WORD, array_size:WORD, exist:BYTE
    .CODE
     EXTRN    C RangeError: Far ; ВНЕШНЯЯ процедура
    PUBLIC C calc
arr    EQU   array[BX]
calc      proc far
ARG L:WORD RETURNS res:WORD
        mov    exist,0 ; ok!
        mov    res,1   ; res = 1;
        XOR    SI,SI   ; j=0;
        XOR    BX,BX   ; регистр-индекс
; Подводим указатель BX к последнему элементу
        mov    DI,array_size
        dec    DI           ; array_size-1
        mov    cx, DI
        JCXZ   @@Exit
@@Begin1:
        inc    BX
        inc    BX
        LOOP   @@Begin1
;------     СОБСТВЕННО РЕШЕНИЕ ЗАДАЧИ:
        mov    cx, array_size
@@Begin:
        mov    DI,arr    ; array[i];
        cmp    DI,0   ;   if (array[i]>0)
        JLE    @@Cont ; НЕТ
;========== array[i]>0 ===========================
        inc    SI       ;   j++;
; сохранение регистров перед вызовом внешней процедуры
        push   ax bx cx dx DI SI
```

```
;  Визуализация вычислений на C++
        call  RangeError C, SI,DI
; восстановление регистров после отработки
; внешней процедуры
        pop   SI DI dx cx bx ax
;res *= array[i];
        mov   ax,res
        MUL   DI   ; <dx:ax>=<ax>*res
        JC     @@ERROR    ; ловим переполнение
        mov    res,ax     ; регистром DX ПРЕНЕБРЕГАЕМ!!!
;if (j==l) return res
        cmp   SI,L
        JE    @@Exit
@@Cont:
;  i--;
        dec   BX
        dec   BX
        LOOP  @@Begin
        cmp   SI,0        ; нашли положительные элементы?
        JG    @@Exit      ; ДА
        mov   exist,-1    ; НЕТ положительных элементов!!!
@@Exit:
        ret
@@ERROR:
; признак ошибки по переполнению умножения
        mov   exist,1
        JMP   @@Exit
calc     endp
end
```

ШАГ 6.

Результаты окончательного тестирования

Конечно, реально отладить этот достаточно сложный алгоритм так быстро НЕ получится. Но, к счастью, мы моделируем обучение не в реальном масштабе времени, поэтому у нас все получается быстро и просто. На самом деле это НЕ так. Но, показав достаточно подробно отладку лабораторной работы № 4, я надеюсь, вы кое-что поняли...

Итак, откроем в программе на C++ соответствующие комментарии и протестируем полученный вариант программы. Посмотрим на наши тесты:

```
======== test #1==============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или     Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 22
=================================================================
              Array:
-58    77     -35    121    -100   62     -5     52     -81    60
-94    99     -43    41     -44    85     -120   54     -87    38
-86       75

Введите количество умножаемых элементов
Значение должно быть от 1 до 22 ===> 2
```

```
C++ function calculation
75*38 = 2850
Asm function calculation
75*38 = 2850

========= test #2===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 50
================================================================
              Array:
 -14  23    -17   118   -11   98    -24   117   -53   24
 -25  58    -27   56    -118  93    -93   50    -62   33
 -84  115   -4    33    -28   102   -23   118   -80   100
 -96  119   -52   24    -3    57    -53   78    -129  86
-112   30   -97  29   -28  13   -66  87   -30  3

Введите количество умножаемых элементов
Значение должно быть от 1 до 50 ===> 5

C++ function calculation
3*87*13*29*30 = 2.95191e+06
Ошибка!!! Результат превышает допустимый диапазон: 65535

Asm function calculation
3*87*13*29
Ассемблер: Произведение до конца вычислить НЕ удалось!!!

========= test #3===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=10000
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 100
================================================================
              Array:
 -58    94      -42     25      -45     18      -110    65      -119
44      -129    117     -14     85      -41     22      -5        13
-99       89
 -81        5       -61        11       -53       103       -89
10      -12         95
 -42       29       -30       122       -11         4       -62
13      -78         32
  -4        86       -78        87       -68       111       -73
40        -2         92
-101       32       -57        81       -53         9        -4
76      -42         22
 -20       86       -49       103       -15       106      -125
22      -29         88
 -50       73      -102        44      -126        77       -47
101       -50        85
-105        0       -32       106       -75        69       -33
1      -92         29
 -13      106      -127        24       -49       108      -107
94      -79        102
Введите количество умножаемых элементов
Значение должно быть от 1 до 100 ===> 2
C++ function calculation
```

```
102*94 = 9588
Asm function calculation
102*94 = 9588

========= test #4===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 35
=================================================================
             Array:
-15         85       -55        55         -8        85       -85
   31       -48         51
-10          0       -47        75       -44          3       -35
   17       -80         38
-92         86       -78       111       -87         24       -45
   106       -89         71
-81       108       -27        74       -36
Введите количество умножаемых элементов
Значение должно быть от 1 до 35 ===> 5
C++ function calculation
74*108*71*106*24 = 1.44355e+09
Ошибка!!! Результат превышает допустимый диапазон: 65535
Asm function calculation
74*108*71
Ассемблер: Произведение до конца вычислить НЕ удалось!!!

========= test #5===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 5
Enter A[0]
Значение должно быть от -32768 до 32767 ===> -23
Enter A[1]
Значение должно быть от -32768 до 32767 ===> 22
Enter A[2]
Значение должно быть от -32768 до 32767 ===> 33
Enter A[3]
Значение должно быть от -32768 до 32767 ===> 222
Enter A[4]
Значение должно быть от -32768 до 32767 ===> -22
=================================================================
             Array:
      -23         22         33       222       -22
Введите количество умножаемых элементов
Значение должно быть от 1 до 5 ===> 3
C++ function calculation
222*33*22 = 161172
Ошибка!!! Результат превышает допустимый диапазон: 65535
Asm function calculation
222*33*22
Ассемблер: Произведение до конца вычислить НЕ удалось!!!
```

```
========= test #6===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 5
Enter A[0]
Значение должно быть от -32768 до 32767 ===> -21
Enter A[1]
Значение должно быть от -32768 до 32767 ===> 0
Enter A[2]
Значение должно быть от -32768 до 32767 ===> -123
Enter A[3]
Значение должно быть от -32768 до 32767 ===> 0
Enter A[4]
Значение должно быть от -32768 до 32767 ===> -21
==============================================================
            Array:
      -21        0     -123         0       -21
Введите количество умножаемых элементов
Значение должно быть от 1 до 5 ===> 3
C++ function calculation
C++: В данном массиве НЕТ положительных элементов!!!
Asm function calculation
ASM: В данном массиве НЕТ положительных элементов!!!
```

Тестирование закончилось успешно. Анализ этих тестов показывает, что **нормально решить нашу задачу, применяя 16-разрядное программирование, НЕЛЬЗЯ**. Очень уж узок при этом получается диапазон допустимых значений! Отметим для себя этот момент — мы к нему еще вернемся и решим эту задачу по-другому...

Можете полюбопытствовать, как эту задачу решил компилятор C++. Не забыли, как это делается? А, если подзабыли, не беда — см. п. 6.1.2.1. Очень полезно учиться и у компиляторов!..

6

Лабораторная работа № 6

Использование цепочечных команд – команд обработки строк (процессор i8086/i286)

Кто думает, что постиг всё,
тот ничего не знает.

Лао-Цзы (IV-III вв. до н.э.)

Цель работы

Применить обработку одномерного целочисленного массива из лабораторной работы № 5, используя в модуле на Ассемблере представление целочисленного массива как строку, содержащую элементы нужной длины, и цепочечные команды. Сравнить реализацию машинного кода ASM-модуля данной работы с кодом лабораторной работы № 5.

Контрольные вопросы

1. Особенности выполнения изучаемых базовых команд процессора i8086/i286.
2. Команды безусловного перехода.
3. Команды условного перехода. Организация разветвлений.
4. Разница в организации условных переходов для знаковых и беззнаковых данных.
5. Команды управления циклом.
6. Основные особенности работы с цепочечными командами: хранение строки-источника и строки-приемника, флаг направления, использование регистров **SI** и **DI**.
7. Целочисленный массив как строка в Ассемблере.
8. Префикс повторения.

9. Соглашения о передаче параметров, принятые для используемого в лабораторной работе алгоритмического языка.
10. Состояние стека при выполнении процедур и функций с параметрами.
11. Ассемблирование и дизассемблирование команд на своих примерах.
12. Понятие о байтах кода операции, способах адресации, регистрах и смещениях.
13. Необходимость в контроле диапазона целочисленных данных при вводе.
14. Диапазон допустимых значений для целочисленных переменных: **char (SHORTINT), unsigned char (BYTE), short int (INTEGER), unsigned short int (WORD), int, unsigned int, unsigned long int (LONGINT)**.

Пример решения типового варианта

Вариант 62 – short int (INTEGER):

Найти произведение последних **L** положительных элементов в массиве **A={a[i]}**.

Длина массива **N=100**. Исходные данные задать самостоятельно, учитывая формат элементов массива **A**.

В программе на С++ (ПАСКАЛЬ) должны быть предусмотрены функции ввода-вывода элементов массива **A** и его обработки. Исходные данные должны вводиться корректно и с проверкой на область допустимых значений. Тип результата определяется из контекста задачи.

Реализация в Borland C++ 5.02 и Turbo Assembler 4.1

Решение

Главную программу для нашего примера оставим такой же, как и в лабораторной работе № 5, – на алгоритмическом языке С++ (Borland С++ 5.02) для 16-разрядной платформы MS DOS.

При выполнении данной лабораторной работы нам понадобятся материалы главы 11, а также глав 5-6, 9. Конечно, здесь очень пригодится и опыт, приобретенный нами при выполнении лабораторных работ № №3-5.

Алгоритмически при решении нашей задачи ничего НЕ изменится, только **представление целочисленного массива и его обработка будут выполнены по-другому.**

ШАГ 1. Представление целочисленного массива с элементамитипа short int в виде строки-источника

Байты в ОЗУ	1	2	3	4	5	6	...
Индекс-переменная (i)	0		1		2		...
Элементы массива (значения)	Array[0]		Array[1]		Array[2]		...
Адреса (смещения)	Array	Array+1	Array+2	Array+3	Array+4	Array+5	...
Индекс-регистр	SI_0	SI_0+1	SI_0+2	SI_0+3	SI_0+4	SI_0+5	...

Для строки-источника в качестве индекс-регистра всегда выбирается регистр **SI**, а доступ к любому элементу одномерного массива **Array [i]** длиной 2 байта подчиняется в Ассемблере той же закономерности, что была изложена в п. 9.4.1:

Array[i] ===> Word PTR[SI_i],

где

$SI_i = SI_0 + 2*i = SI_{i-1} + 2$;
SI_0 — адрес начала массива **Array**;

i=0,...,n-1; **n** — длина массива **Array**.

Отличие от лабораторной работы № 5:

- **индексный регистр — вполне определенный** (в нашем случае регистр SI);
- **в индексном регистре всегда находится АДРЕС элемента массива;**
- **в силу последнего замечания способ адресации ДРУГОЙ — косвенный по индексу.** Для нашей задачи Array[i] ===> WORD PTR [SI_i]. Эта запись означает, что нужно взять содержимое слова из области памяти по адресу, хранящемуся в регистре SI_i.

ШАГ 2.
ASM-процедура calc

Выделим жирным шрифтом строки ASM-кода, специфичные для цепочечных команд. Регистр **BX**, который в лабораторной работе № 5 использовался как индексный, в этой работе будет использоваться в качестве счетчика нужных нам положительных элементов. Регистр **DI** используем по своему усмотрению, поскольку со строкой-приемником мы в данном случае НЕ работаем.

Исходный текст модуля Func6.asm

```
title Lab_6 (CopyRight by Голубь Н.Г., 2001)
    .MODEL LARGE, C
    LOCALS @@
    DATASEG
; ГЛОБАЛЬНЫЕ переменные
EXTRN C array:WORD,array_size:WORD,exist:BYTE

   CODESEG
   EXTRN  C RangeError: Far ; ВНЕШНЯЯ процедура
    PUBLIC C calc
calc     proc far
ARG L:WORD RETURNS res:WORD
    push DS
    pop  ES            ; ES=DS
    CLD                ; Направление обработки =====>
    LEA  SI,array      ; Адрес начала массива
    mov  exist,0       ; ok!
    mov  res,1         ; res = 1;
    XOR  BX,BX         ; j=0;
```

```
; Подводим указатель SI к последнему элементу
   mov  DI,array_size
   dec  DI     ; array_size-1
   mov  cx, DI
   JCXZ @@Exit
REP  LODSW
;———————— СОБСТВЕННО РЕШЕНИЕ ЗАДАЧИ:
   mov  cx, array_size
@@Begin:

   cmp  WORD PTR[SI],0 ; array[i]~0
   JLE  @@Cont       ; if (array[i]<=0)
;========== array[i]>0 ===========================
   inc  BX    ; j++;
; сохранение регистров перед вызовом внешней процедуры
   push bx cx SI
; Визуализация вычислений на C++
   call RangeError C, BX,WORD PTR[SI]
; восстановление регистров после отработки
; внешней процедуры
   pop  SI cx bx
;res *= array[i];
   mov  ax,res
   MUL  WORD PTR[SI] ; <dx:ax>=res*array[i]
   JC   @@ERROR ; ловим переполнение
   mov  res,ax  ; регистром DX ПРЕНЕБРЕГАЕМ!!!
;if (j==1) return res
   cmp  BX,L
   JE   @@Exit
@@Cont:
; i--;
   dec  SI
   dec  SI
   LOOP @@Begin
   cmp  BX,0   ; нашли положительные элементы?
   JG   @@Exit  ; ДА
   mov  exist,-1 ; НЕТ положительных элементов!!!
@@Exit:
   ret
@@ERROR:
; признак ошибки по переполнению умножения
   mov  exist,1
   JMP  @@Exit
calc  endp
end
```

ШАГ 3.

Тестирование программы

```
========= test #1===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или   Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 25
================================================================
      Array:
-21    65   -115    99    -3    69   -128    31    -19     8
-16     3    -89    109    -9    117    -72    112    -89    38
-45    15   -114    108   -107
Введите количество умножаемых элементов
Значение должно быть от 1 до 25 ===> 3
C++ function calculation
108*15*38 = 61560
Asm function calculation
108*15*38 = 61560
========= test #2===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или   Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 5
Enter A[0]
Значение должно быть от -32768 до 32767 ===> -4213
Enter A[1]
Значение должно быть от -32768 до 32767 ===> 0
Enter A[2]
Значение должно быть от -32768 до 32767 ===> 22
Enter A[3]
Значение должно быть от -32768 до 32767 ===> 33
Enter A[4]
Значение должно быть от -32768 до 32767 ===> 5
================================================================
      Array:
  -4213     0    22    33     5
Введите количество умножаемых элементов
Значение должно быть от 1 до 5 ===> 4
C++ function calculation
5*33*22 = 3630
Asm function calculation
5*33*22 = 3630
========= test #3===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или   Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 100
================================================================
      Array:
 -80    77    -76    13    -67    54    -15    20   -101    65
-124    54    -54    20    -77    98    -80    78    -97    23
 -10    41    -97    59    -49    60    -38    69    -60    52
 -19   115   -102   105    -53    87    -31    21   -116    91
 -65    49    -36    24   -102    31    -71    29    -93    44
 -54     0    -43    50    -23    91   -109    43   -102     0
```

```
  -31    47    -67    33    -42    22    -4    28    -44    58
  -32    47    -79    21   -105    88   -66    65    -59     3
  -25     0    -38   114    -32   100   -15    73   -111    48
 -101   111    -64    29   -108   128  -123    13    -52   120
Введите количество умножаемых элементов
Значение должно быть от 1 до 100 ===> 5
C++ function calculation
120*13*128*29*111 = 6.4277e+08
Ошибка!!! Результат превышает допустимый диапазон: 65535
Asm function calculation
120*13*128
Ассемблер: Произведение до конца вычислить НЕ удалось!!!
```

Тесты дали нормальные результаты.

ШАГ 4.
Сравнение реализаций машинного кода ASM-модулей

Для этого нужно проанализировать концовки файлов **Func6.lst** и **Func.lst**.

Фрагмент файла Func6.lst

```
...........................................................
64  005E EB F7                  JMP  @@Exit
65  0060                    calc   endp
66                          end
...........................................................
DGROUP                          Group
  _DATA                         16  0000  Word      Public  DATA
FUNC6_TEXT                      16  0060  Word      Public  CODE
```

Фрагмент файла Func.lst

```
...........................................................
66  0065 EB F7                  JMP  @@Exit
67  0067                    calc   endp
68                          end
...........................................................
DGROUP                          Group
  _DATA                         16  0000  Word      Public  DATA
FUNC_TEXT                       16  0067  Word      Public  CODE
```

Длина машинного кода модуля **Func6.asm** получилась равной **60h** (96 байт), а модуля **Func.asm** — **67h** (103 байта). Выигрыш получился в 7 байт. В общем, не очень много, но и программка сама не большая... Попробуйте сделать ЛУЧШЕ!

 **ЗАМЕЧАНИЕ**

В данной лабораторной работе мы употребили не столько цепочечные команды (из них применена всего одна: **REP LODSW**), сколько использовали представление целочисленного массива в виде строки, содержащей данные длиной 2 байта. Соответственно немного изменилась и обработка такого массива.



Лабораторная работа № 7

Особенности 32-разрядного программирования (процессор i386/i486/Pentium)

Долог путь поучений, короток и успешен путь примеров.

Сенека Луций Анней мл.
(ок. 4 г. до н.э. — 65 г. н.э.)

Цель работы

Вычислить заданное в лабораторной работе № 4 **условное** выражение для **знаковых целочисленных 32-разрядных данных**, используя команды сравнения, условного и безусловного переходов. Результат **X** — тоже **целочисленный**. Исходные данные должны вводиться корректно (с проверкой на область допустимых значений). Результат также должен быть проверен на область допустимых значений. Данные должны передаваться в подпрограммы (функции) как параметры.

Порядок работы

1) внимательно изучить свой вариант условного арифметического выражения;
2) написать на базовом алгоритмическом языке программу ввода исходных данных (с контролем допустимого диапазона), вычисления условного арифметического выражения и вывода полученного результата;
3) написать модуль вычисления условного арифметического выражения на языке Ассемблера;
4) встроить вызов этого модуля в программу на базовом алгоритмическом языке;
5) произвести тестовые проверки, отметить нормальные и аномальные результаты, сделать анализ результатов.

Контрольные вопросы

1. Особенности выполнения изучаемых команд процессора **i386/i486/Pentium**.
2. Команды безусловного перехода.
3. Команды условного перехода. Организация разветвлений.
4. Соглашения о передаче параметров, принятые для используемого в лабораторной работе алгоритмического языка.
5. Состояние стека при выполнении процедур и функций с параметрами.
6. Ассемблирование и дизассемблирование команд на своих примерах.
7. Режимы адресации в 32-разрядном программировании.
8. Понятие о байтах кода операции, способа адресации, SIB, регистрах и смещениях.
9. Необходимость в контроле диапазона целочисленных данных при вводе.
10. Диапазон допустимых значений для целочисленных переменных при 32-разрядном программировании.

Пример решения типового варианта

Будем решать тот же 62 вариант, что в лабораторной работе № 4:

$$X = \begin{cases} b/a - 4, & \text{если} \quad a < b, \\ 25, & \text{если} \quad a = b, \\ (a^3 - 5)/b, & \text{если} \quad a > b; \end{cases}$$

для **знаковых целочисленных 32-разрядных данных**, используя команды сравнения, условного и безусловного переходов. Результат X — тоже **целочисленный**. Исходные данные должны вводиться корректно (с проверкой на область допустимых значений). Результат также должен быть проверен на область допустимых значений. Данные должны передаваться в подпрограммы (функции) как параметры.

Реализация в Borland C++ 5.02 и Turbo Assembler 5.3

Решение

Сделаем для нашего примера главную программу тоже на алгоритмическом языке C++ (Borland C++ 5.02), но **платформа разработки будет уже другая: Application.exe | Win32 Console**. Поэтому и компилятор Ассемблера тоже должен быть 32-разрядным: **tasm32.exe** версии 5.x. Принимаем более продвинутый компилятор версии 5.3, входящий в стандартную поставку Borland C++ Builder 5.0.

При выполнении данной лабораторной работы нам понадобятся в основном материалы главы 12 и глав 5-6, 9. Кроме того, здесь будет полезен и опыт, приобретенный нами при выполнении лабораторной работы № 4.

Проиллюстрируем наши действия по шагам.

ШАГ 0.
Анализ особенностей задачи

1. Алгоритм вычисления условного выражения пока (до тестирования) оставим такой же, как и в лабораторной работе № 4 (мы с ним достаточно долго провозились).
2. Это позволит нам сделать модуль на 32-разрядном Ассемблере простой заменой 16-разрядных регистров на 32-разрядные. Заменим и команду распространения знака **CWD** на **CDQ**. Кроме того, смещения при условных переходах тоже становятся по умолчанию 32-разрядными, поэтому всюду, где это возможно, поставим SHORT. Тогда смещения станут опять 8-битными, и мы на каждой команде сэкономим по 4 байта.
3. Изменим главную программу на C++.
   - Удалим все фрагменты исходного кода, связанные с беззнаковыми данными и организацией меню.
   - Параметризованных функций НЕ будет, поскольку мы собираемся работать только с 32-разрядными данными типа **int**.
   - Разрешим и Ассемблеру (наряду с C++) тоже делать вычисления, не считаясь с угрозой "ДЕЛЕНИЕ НА НОЛЬ". Посмотрим, к чему это приведет.
   - Анализ битов **ios::X** на этапе ввода сделать для проекта **Win32 Console** НЕ удастся, поскольку этот анализ не поддерживается самим компилятором Borland C++ 5.02 (программа на этапе выполнения зацикливается).
   - В связи с тем, что переменная **int** стала 32-разрядной, кардинально изменится функция **test** — проверка на допустимый диапазон.

ШАГ 1.
Главная программа на C++

Так же, как и в лабораторной работе № 4, решим и отладим сначала нашу задачу на C++. Выделим **жирным шрифтом** наиболее значимые места исходного кода, которые изменились по сравнению с лабораторной работой № 4.

Исходный текст программы Lab4.cpp

```
/* Lab4.cpp                Borland C++ 5.02
        Проект !!!!!! Application.exe === > Win32 Console !!!!!!
                    (c) Copyright 1998-2001 by Голубь Н.Г.
                     Лабораторная работа # 4-32.
*/
#include <limits.h>
#include <iostream.h>
#include <conio.h>
const char* CONTINUE = "\n\n Press any key to continue...";
// Превышение диапазона для INT при вычислении ВЫРАЖЕНИЯ
 char* ERROR_RANGE_EXPRESSION=
                        "\nC++: !!! ERROR RANGE EXPRESSION. Value = ";
// ГЛОБАЛЬНАЯ переменная — для Ассемблера
 unsigned char f;
//!!!!!!!!!!! Описание внешней ASM-процедуры !!!!!!!!!!!
extern "C"
```

```
{   int lab4l_32 (int a, int b);
}
//!!!!!!!!!!!!!!!!!!!!!!!!!!!
int test (double x, long CONST_MIN=LONG_MIN, long CONST_MAX=LONG_MAX)
{
     if (x > CONST_MIN && x < CONST_MAX)
        return 1;  // Все нормально!
    else
      {  cout<<ERROR_RANGE_EXPRESSION << x << endl;
         return 0;
        }
}
void title(int& t)
{
    clrscr();
    cout << "\n================== test #" << t++ << " ====================\n";
    cout <<"\n            Variant 62:      "<< endl;
     cout << "             b/a-4     , a<b "<< endl;
     cout << "        y=   25        , a=b "<< endl;
     cout << "            (a^3-5)/b , a>b "<< endl;
           cout << "Exit — (Ctrl-c)"<< endl;
}
void PressAnyKey()
{ cout <<  CONTINUE;
   getch();
}
int func(int a, int b)
 {
    double x;
     if (a<b)
     {
      cout << "            b/a-4     ; " << a << " <  " << b << endl;
      if (a) x=b/a-4;
       else
         {
          cout << "C++: ERROR!!! Divide by ZERO!!!";
          return 1;
         }
     }
      else
        if (a>b)
         {
          cout << "            (a^3-5)/b ; " << a << " >  " << b << endl;
          if(b) x=((double)a*a*a-5.)/b; //!!!!!!!!!!
          else
               {
                cout << "C++: ERROR!!! Divide by ZERO!!!";
                return 1;
               }
             }
        else
         { cout << a << " =  " << b << endl;
           x=25;
          }
```

```
// Проверка на ДОПУСТИМЫЙ диапазон результата
    if (test(x)) cout<<"CPP:   Result =  " << (int)x << endl;
    return 0;// Ok!
  }
void ProcInt(double a, double b, int y)
{ // Ввод с контролем допустимого диапазона
   do
     {
      cout << "a (int)  "; cin >> a;
     } while (!test(a));
   do
     {
      cout << "b (int)  "; cin >> b;
     } while (!test(b));
// Вызов С++ функции
   func((int)a,(int)b);
/* Вызов процедуры-функции на ассемблере пока закомментируем!!!!!!!
   y=lab4l_32((int)a,(int)b);  // Вызов процедуры на ассемблере
   if (!f) cout <<"\nASM:   Result =  " << y << endl;
   else if (f==1) cout << "\nASM: ERROR!!! Divide by ZERO!!!";
   else cout << "\nASM: Sorry!!! Overflow at 32-digit multiplication!!!";
*/
}
int main()
{ double a,b;
   int y,t=1;
   for(;;)
    { title(t);
      ProcInt(a,b,y);
      PressAnyKey();
    }
}
```

ШАГ 2.

Тестирование программы — вариант 1

```
==================== test #1 =======================
               Variant 62:
              b/a-4       , a<b
        y=    25          , a=b
              (a^3-5)/b   , a>b
Exit — (Ctrl-c)
a (int)   111
b (int)   111
111 =   111
CPP:    Result =   25
 Press any key to continue...
==================== test #2 =======================
               Variant 62:
              b/a-4       , a<b
        y=    25          , a=b
              (a^3-5)/b   , a>b
Exit — (Ctrl-c)
a (int)   111111
b (int)   11111111
               b/a-4       ; 111111 <   11111111
```

```
CPP:    Result =  96
 Press any key to continue...
===================== test #3 =======================
              Variant 62:
             b/a-4      , a<b
        y=   25         , a=b
             (a^3-5)/b  , a>b
Exit - (Ctrl-c)
a (int)  1111
b (int)  1
             (a^3-5)/b ; 1111 >  1
CPP:    Result =  1371330626
 Press any key to continue...
===================== test #4 =======================
              Variant 62:
             b/a-4      , a<b
        y=   25         , a=b
             (a^3-5)/b  , a>b
Exit - (Ctrl-c)
a (int)  12345
b (int)  -1234567890
             (a^3-5)/b ; 12345 >  -1234567890
CPP:    Result =  -1523
 Press any key to continue...
===================== test #5 =======================
              Variant 62:
             b/a-4      , a<b
        y=   25         , a=b
             (a^3-5)/b  , a>b
Exit - (Ctrl-c)
a (int)  1234567
b (int)  0
             (a^3-5)/b ; 1234567 >  0
C++: ERROR!!! Divide by ZERO!!!
 Press any key to continue...
===================== test #6 =======================
              Variant 62:
             b/a-4      , a<b
        y=   25         , a=b
             (a^3-5)/b  , a>b
Exit - (Ctrl-c)
a (int)  0
b (int)  1234567890
             b/a-4     ; 0 <  1234567890
C++: ERROR!!! Divide by ZERO!!!
 Press any key to continue...
===================== test #7 =======================
              Variant 62:
             b/a-4      , a<b
        y=   25         , a=b
             (a^3-5)/b  , a>b
Exit - (Ctrl-c)
a (int)  8672247896
C++: !!! ERROR RANGE EXPRESSION. Value = 8.67225e+09
a (int)  4564675
b (int)  -6345857
```

```
              (a^3-5)/b ; 4564675 >  -6345857
C++: !!! ERROR RANGE EXPRESSION. Value = -1.49878e+13
 Press any key to continue...
====================== test #8 ========================
              Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
              (a^3-5)/b  , a>b
Exit — (Ctrl-c)
a (int)  -4232345
b (int)  24342342453
C++: !!! ERROR RANGE EXPRESSION. Value = 2.43423e+10
b (int)  453453543
              b/a-4      ; -4232345 <  453453543
CPP:    Result =  -111
 Press any key to continue...
====================== test #9 ========================
              Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
              (a^3-5)/b  , a>b
Exit — (Ctrl-c)
a (int)  1234
b (int)  2
              (a^3-5)/b ; 1234 >  2
CPP:    Result =  939540449
 Press any key to continue...
====================== test #10 ========================
              Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
              (a^3-5)/b  , a>b
Exit — (Ctrl-c)
a (int)  1234
b (int)  1
              (a^3-5)/b ; 1234 >  1
CPP:    Result =  1879080899
 Press any key to continue...
```

Тест работает нормально.

Шаг 3.
ASM-процедура вычисления условного арифметического выражения для 32-битных знаковых данных

```
title Lab_4-32 (CopyRight by Голубь Н.Г., 1998, 2001)
.386
model FLAT,C
LOCALS @@
;             b/a-4          , a<b
;       y=    25             , a=b
;            (a^3-5)/b       , a>b
CODESEG
EXTRN  C f:BYTE  ; Флажок "Деление на ноль": f=1
; Флажок "Переполнение при 32-разрядном умножении" f=2
public C lab4l_32
lab4l_32 proc   ; Данные знаковые!!!
ARG a:DWORD,b:DWORD RETURNS y:DWORD
    ;================ Начало процедуры =========
    mov   f,0                        ;ok!
    mov   Eax,a                      ;<Eax>=a
    mov   Ebx,b                      ;<Ebx>=b
    cmp   Eax,Ebx                    ; сравнение a~b
    jg    SHORT @@2                  ;a>b goto @@2
    jl    SHORT @@1                  ;a<b goto @@1
    ;================== a = b ==================
    mov   Eax,25
    jmp   short @@Exit
    ;================== a < b ==================
@@1:
    cmp   Eax,0                      ; сравнение a~0
    JE    SHORT @@ERROR              ; a=0 goto @@ERROR
    mov   Eax,Ebx                    ; <Eax>=b
    CDQ                              ; <Eax> ===> <Edx:Eax>
    IDIV  a                          ; <Eax>=<Edx:Eax>/a
    SUB   Eax,4                      ; <ax>=b/a-4
    jmp   short @@Exit
    ;==================a > b ==================
@@2:
    cmp   Ebx,0                      ; сравнение b~0
    JE    SHORT @@ERROR              ; a=0 goto @@ERROR
    Imul  a                          ;<Edx:Eax>=a*a
    JO    SHORT @@ERRORm             ;Переполнение при знаковом умножении
    Imul  a                          ;<Edx:Eax>=a*a*a
    add   Eax,-5                     ;   <Eax>=<Eax>-5  (мл.часть)
    adc   Edx,0FFFFFFFFh             ; !!!!!!!!  <Edx>=<Edx> -(ст.часть числа -5)
    Idiv  Ebx                        ; <Eax>=(a*a*a-5)/b
    jmp   short @@Exit
    ;=============== Деление на ноль =========
@@ERROR:
    mov   f,1
    ret
    ;=============== Переполнение при 32-разрядном умножении =========
@@ERRORm:
    mov   f,2
    ret
```

```
    ;============== Получение результата =========
@@Exit:
    mov y,Eax
    ret
    ;=========== Конец процедуры ================
lab4l_32 endp
end
```

ШАГ 4.
Тестирование и анализ программы с ASM-модулем

Откроем в программе на C++ соответствующие комментарии и протестируем полученный вариант программы.

```
====================== test #1 ========================
               Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
              (a^3-5)/b  , a>b
Exit — (Ctrl-c)
a (int)  1111
b (int)  1110
              (a^3-5)/b ; 1111 >  1110
CPP:    Result =  1235432
ASM:    Result =  1235432
 Press any key to continue...
====================== test #2 ========================
               Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
              (a^3-5)/b  , a>b
Exit — (Ctrl-c)
a (int)  0
b (int)  0
0 =  0
CPP:    Result =  25
ASM:    Result =  25
 Press any key to continue...
====================== test #3 ========================
               Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
              (a^3-5)/b  , a>b
Exit — (Ctrl-c)
a (int)  100
b (int)  1
              (a^3-5)/b ; 100 >  1
CPP:    Result =  999995
ASM:    Result =  999995
 Press any key to continue...
====================== test #4 ========================
               Variant 62:
              b/a-4      , a<b
        y=    25         , a=b
              (a^3-5)/b  , a>b
```

```
Exit — (Ctrl-c)
a (int)  200
b (int)  -30000
         (a^3-5)/b ; 200 >  -30000
CPP:   Result =  -266
ASM:   Result =  -266
 Press any key to continue...
===================== test #5 =======================
          Variant 62:
         b/a-4     , a<b
     y=  25        , a=b
         (a^3-5)/b , a>b
Exit — (Ctrl-c)
a (int)  2222
b (int)  2221
         (a^3-5)/b ; 2222 >  2221
CPP:   Result =  4939506
ASM:   Result =  4939506
 Press any key to continue...
===================== test #6 =======================
          Variant 62:
         b/a-4     , a<b
     y=  25        , a=b
         (a^3-5)/b , a>b
Exit — (Ctrl-c)
a (int)  22222
b (int)  22221
         (a^3-5)/b ; 22222 >  22221
CPP:   Result =  493839506
ASM:   Result =  493839506
 Press any key to continue...
===================== test #7 =======================
          Variant 62:
         b/a-4     , a<b
     y=  25        , a=b
         (a^3-5)/b , a>b
Exit — (Ctrl-c)
a (int)  55555
b (int)  -999
         (a^3-5)/b ; 55555 >  -999
C++: !!! ERROR RANGE EXPRESSION. Value = -1.71634e+11
ASM: Sorry!!! Overflow at 32-digit multiplication!!!
 Press any key to continue...
===================== test #8 =======================
          Variant 62:
         b/a-4     , a<b
     y=  25        , a=b
         (a^3-5)/b , a>b
Exit — (Ctrl-c)
a (int)  1
b (int)  111111111
         b/a-4     ; 1 <  111111111
CPP:   Result =  111111107
ASM:   Result =  111111107
 Press any key to continue...
```

```
===================== test #9 =======================
              Variant 62:
             b/a-4      , a<b
        y=   25         , a=b
             (a^3-5)/b  , a>b
Exit - (Ctrl-c)
a (int)  -3333
b (int)  -33333
             (a^3-5)/b ; -3333 >  -33333
CPP:   Result =  1110788
ASM:   Result =  1110788
 Press any key to continue...
===================== test #10 ======================
              Variant 62:
             b/a-4      , a<b
        y=   25         , a=b
             (a^3-5)/b  , a>b
Exit - (Ctrl-c)
a (int)  11111
b (int)  111
             (a^3-5)/b ; 11111 >  111
C++: !!! ERROR RANGE EXPRESSION. Value = 1.23577e+10
????????? Деление на НОЛЬ в Ассемблере ??????????
 Press any key to continue...
===================== test #11 ======================
              Variant 62:
             b/a-4      , a<b
        y=   25         , a=b
             (a^3-5)/b  , a>b
Exit - (Ctrl-c)
a (int)  2314423
b (int)  -4141
             (a^3-5)/b ; 2314423 >  -4141
C++: !!! ERROR RANGE EXPRESSION. Value = -2.9938e+15
ASM: Sorry!!! Overflow at 32-digit multiplication!!!
 Press any key to continue...
```

Все шло хорошо, пока **test #10** не дал особую ситуацию "Деление на НОЛЬ" при вычислениях в Ассемблере. По умолчанию **для 32-разрядного консольного режима в среде Borland C++ 5.02** при возникновении особой ситуации автоматически вызывается ОЧЕНЬ неплохой отладчик, и можно эту ситуацию детально проанализировать — см. рис. Л7.1

Хотя в нашем случае знаменатель явно НЕ равен НУЛЮ (**b=111**), но зато числитель ОЧЕНЬ большой и при делении результат НЕ поместился в регистр **EAX**, что и вызвало ситуацию "Деление на НОЛЬ" — см. п. 5.2.4. Будем контролировать в Ассемблере переполнение и при втором умножении. Это сузит нам диапазон решения, но зато избавит от неприятной ситуации. **Нормально решить данную проблему в Ассемблере можно только с помощью СОПРОЦЕССОРА**. Вот почему в С++ мы сделали ЯВНОЕ преобразование при вычислении x=((**double**)a*a*a-5.)/b.

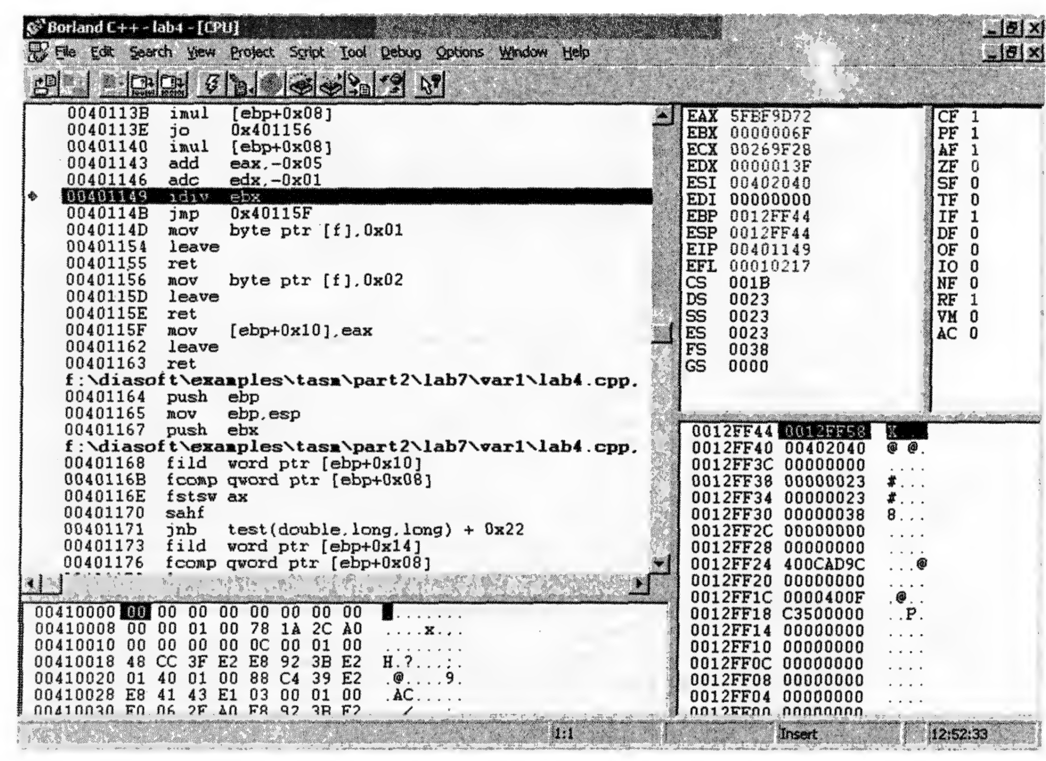


РИС. Л7-1. *Отладчик Borland C++ 5.02*

ШАГ 5.
Исправление модуля на Ассемблере и программы на С++

Окончательный вариант модуля Lab_4.asm

```
title Lab_4-32 (CopyRight by Голубь Н.Г., 1998, 2001)
.386
model   FLAT,C
LOCALS  @@
;                    b/a-4       , a<b
;           y=      25            , a=b
;                  (a^3-5)/b   , a>b
CODESEG
EXTRN   C f:BYTE   ; Флажок "Деление на ноль": f=1
; Флажок "Переполнение при 32-разрядном умножении" f=2
; "Переполнение при 32-разрядном умножении еще раз" f=3
public C lab4I_32
lab4I_32 proc    ; Данные знаковые!!!
ARG a:DWORD,b:DWORD RETURNS y:DWORD
  ;================ Начало процедуры ========
  mov f,0       ;ok!
  mov Eax,a     ;<Eax>=a
  mov Ebx,b     ;<Ebx>=b
  cmp Eax,Ebx   ; сравнение a~b
```

```
  jg   SHORT @@2          ;a>b goto @@2
  jl   SHORT @@1          ;a<b goto @@1
  ;================== a = b ==================
  mov  Eax,25
  jmp  short @@Exit
  ;================== a < b ==================
@@1:
  cmp  Eax,0       ; сравнение a~0
  JE   SHORT @@ERROR   ; a=0 goto @@ERROR
  mov  Eax,Ebx      ; <Eax>=b
  CDQ              ; <Eax> ===> <Edx:Eax>
  IDIV a           ; <Eax>=<Edx:Eax>/a
  SUB  Eax,4       ; <ax>=b/a-4
  jmp  short @@Exit
  ;==================a > b ==================
@@2:
  cmp  Ebx,0       ; сравнение b~0
  JE   SHORT @@ERROR   ; a=0 goto @@ERROR
  Imul a          ;<Edx:Eax>=a*a
;Переполнение при знаковом умножении
  JO   SHORT @@ERRORm
  Imul a          ;<Edx:Eax>=a*a*a
;Переполнение при знаковом умножении еще раз
;       - угроза ситуации "Деление на ноль"
  JO   SHORT @@ERRORm2
  add  Eax,-5              ;    <Eax>=<Eax>-5   (мл.часть)
; !!!!!!!!    <Edx>=<Edx> -(ст.часть числа -5)
  adc   Edx,0FFFFFFFFh
  Idiv Ebx         ; <Eax>=(a*a*a-5)/b
  jmp  short @@Exit
  ;=============== Деление на ноль =========
@@ERROR:
     mov  f,1
     ret
;======= Переполнение при 32-разрядном умножении ====
@@ERRORm:
     mov  f,2
     ret
;=============== Переполнение при 32-разрядном
;                умножении еще раз =========
@@ERRORm2:
     mov  f,3
     ret
  ;=============== Получение результата =========
@@Exit:
     mov  y,Eax
     ret
  ;=========== Конец процедуры ================
lab4I_32  endp
end
```

Изменения в программе на C++ (файл Lab4.cpp)

```
// Вызов C++ функции
func((int)a,(int)b);
y=lab4I_32((int)a,(int)b);  // Вызов процедуры на ассемблере
if (!f) cout <<"\nASM:  Result =  " << y << endl;
else if (f==1) cout << "\nASM: ERROR!!! Divide by ZERO!!!";
else if (f==2)cout << "\nASM: Sorry!!! Overflow at 32-digit multiplication!!!";
// Возможна ситуация "ДЕЛЕНИЕ НА НОЛЬ"
else cout << "\nASM: Sorry!!! The situation is possible: DIVIDE BY ZERO!!!";
```

Шаг 6.

Тестирование исправленного варианта

```
====================== test #1 ========================
              Variant 62:
              b/a-4      , a<b
        y=   25          , a=b
             (a^3-5)/b  , a>b
Exit - (Ctrl-c)
a (int)  1111
b (int)  111
             (a^3-5)/b ; 1111 >  111
CPP:   Result =  12354329
ASM:   Result =  12354329
 Press any key to continue...
====================== test #2 ========================
              Variant 62:
             b/a-4      , a<b
        y=   25          , a=b
             (a^3-5)/b  , a>b
Exit - (Ctrl-c)
a (int)  11111
b (int)  111
             (a^3-5)/b ; 11111 >  111
C++: !!! ERROR RANGE EXPRESSION. Value = 1.23577e+10
ASM: Sorry!!! The situation is possible: DIVIDE BY ZERO!!!
 Press any key to continue...
====================== test #3 ========================
              Variant 62:
              b/a-4      , a<b
        y=   25          , a=b
             (a^3-5)/b  , a>b
Exit - (Ctrl-c)
a (int)  11111
b (int)  11110
             (a^3-5)/b ; 11111 >  11110
CPP:   Result =  123465432
ASM: Sorry!!! The situation is possible: DIVIDE BY ZERO!!!
 Press any key to continue...
====================== test #4 ========================
              Variant 62:
             b/a-4      , a<b
        y=   25          , a=b
             (a^3-5)/b  , a>b
```

```
Exit – (Ctrl-c)
a (int)  -111111
b (int)  11111111
           b/a-4     ; -111111 <  11111111
CPP:   Result =  -104
ASM:   Result =  -104
 Press any key to continue...
===================== test #5 ========================
           Variant 62:
           b/a-4      , a<b
      y=   25         , a=b
           (a^3-5)/b  , a>b
Exit – (Ctrl-c)
a (int)  1111111
b (int)  0
          (a^3-5)/b ; 1111111 >  0
C++: ERROR!!! Divide by ZERO!!!
ASM: ERROR!!! Divide by ZERO!!!
 Press any key to continue...
===================== test #6 ========================
           Variant 62:
           b/a-4      , a<b
      y=   25         , a=b
          (a^3-5)/b , a>b
Exit – (Ctrl-c)
a (int)  0
b (int)  11111222
           b/a-4     ; 0 <  11111222
C++: ERROR!!! Divide by ZERO!!!
ASM: ERROR!!! Divide by ZERO!!!
 Press any key to continue...
===================== test #7 ========================
           Variant 62:
          b/a-4       , a<b
      y=  25          , a=b
          (a^3-5)/b  , a>b
Exit – (Ctrl-c)
a (int)  324421354543
C++: !!! ERROR RANGE EXPRESSION. Value = 3.24421e+11
a (int)  4242135
b (int)  -44213424525
C++: !!! ERROR RANGE EXPRESSION. Value = -4.42134e+10
b (int)  141423
          (a^3-5)/b ; 4242135 >  141423
C++: !!! ERROR RANGE EXPRESSION. Value = 5.39801e+14
ASM: Sorry!!! Overflow at 32-digit multiplication!!!
 Press any key to continue...
```

В тестовом примере #3 функция на С++ дала нормальный результат, но получить его в Ассемблере при 32-разрядном целочисленном программировании НЕВОЗМОЖНО. Нужен, как уже указывалось, СОПРОЦЕССОР.

8

Лабораторная работа № 8

Вычисление арифметических выражений и трансцендентных функций (сопроцессор ix87)

Даже маленькая практика стоит большой теории.

Закон Букера
(Артур Блох "Закон Мерфи")

Цель работы

Вычислить заданное смешанное арифметическое выражение для данных в форматах **float** (SINGLE — переменные **a**, **b**) и **int** (INTEGER — остальные переменные), используя арифметические операции сопроцессора **Intel x87: FADD, FADDP, FIADD, FSUB, FSUBP, FISUB, FSUBR, FSUBRP, FISUBR, FMUL, FMULP, FIMUL, FDIV, FDIVP, FIDIV, FDIVR, FDIVRP, FIDIVR, FSQRT, FPREM, FRNDINT, FXTRACT, FABS, FCHS, F2XM1, FYL2X, FYL2XP1, FPTAN, FPATAN, FLDZ, FLD1, FLDPI, FLDL2T, FLDL2E, FLDLG2, FLDLN2, FILD, FIST, FISTP, FLD, FST, FSTP, FXCH**. Исходные данные задать самостоятельно.

В программе на С++ (ПАСКАЛЬ) должны быть предусмотрены функции ввода-вывода. Исходные данные должны вводиться корректно и с проверкой на область допустимых значений. Тип результата — вещественный и определяется из контекста задачи.

Порядок работы

1) внимательно изучить свой вариант смешанного арифметического выражения;

2) написать модуль вычисления арифметического выражения на языке Ассемблера;

3) написать на базовом алгоритмическом языке программу корректного ввода исходных данных (с контролем допустимого диапазона), вычисления арифметического выражения и вывода полученного результата;

4) встроить вызов этого модуля в программу на базовом алгоритмическом языке;

5) произвести тестовые проверки, отметить нормальные и аномальные результаты, сделать анализ результатов.

Варианты

1) $$\frac{2*c - d + \sqrt{23*a}}{\frac{a}{4} - 1}$$

2) $$\frac{c + 4*d - \sqrt{123*c}}{1 - \frac{a}{2}}$$

3) $$\frac{-2*c + d*82}{tg(\frac{a}{4} - 1)}$$

4) $$\frac{lg(2*c-a) + d - 152}{\frac{a}{4} + c}$$

5) $$\frac{tg(a+c/4) - 12*d}{a*b - 1}$$

6) $$\frac{-2*c - sin(a/d) + 53}{\frac{a}{4} - b}$$

7) $$\frac{2*c - lg(d/4)}{a*a - 1}$$

8) $$\frac{tg(c) - d*23}{2*a - 1}$$

9) $$\frac{2*c - d/23}{ln(1 - \frac{a}{4})}$$

10) $$\frac{4*c + d - 1}{c - tg\frac{a}{2}}$$

11) $$\frac{2*c - d*\sqrt{\frac{42}{d}}}{c + a - 1}$$

12) $$\frac{\sqrt{\frac{25}{c}} - d + 2}{d + a - 1}$$

13) $\dfrac{\text{arctg(c - d/2)}}{\text{2*a - 1}}$

14) $\dfrac{\text{4*lg (c) - d/2 + 23}}{\text{a*a - 1}}$

15) $\dfrac{\text{c*tg(b + 23)}}{\text{a/2 -4*d - 1}}$

16) $\dfrac{\text{c/d + ln(3*a/2)}}{\text{c - a + 1}}$

17) $\dfrac{\text{2*c + lg(d)*51}}{\text{d - a - 1}}$

18) $\dfrac{\text{2*c + ln(d/4) + 23}}{\text{a*a - 1}}$

19) $\dfrac{\text{42*c - d/2 + 1}}{\text{a*a - ln(b-5)}}$

20) $\dfrac{\text{arctg(2*c)/d + 2}}{\text{d - a - 1}}$

21) $\dfrac{\text{arctg(12/c) + 73}}{\text{a*a - 1}}$

22) $\dfrac{\text{2*c/a - d*d}}{\text{d + tg(a - 1)}}$

23) $\dfrac{\sqrt{\frac{53}{a}} + d - 4*a}{1 + a*c}$

24) $\dfrac{\sqrt{15*a} + b - \frac{a}{4}}{c*d - 1}$

25) $\dfrac{\text{-25/a + c - tg(b)}}{\text{1 + c*d/2}}$

26) $\dfrac{\text{lg(4*a - 1) + b/2}}{\text{d*c - 5}}$

27) $\dfrac{\text{8*lg(b + 1) - c}}{\text{a/2 + d*c}}$

28) $\dfrac{\text{4*a - ln(b - 1)}}{\text{c/d + 18}}$

29) $\dfrac{\text{arctg(4*b)/c - 1}}{\text{12 + a - b}}$

30) $\dfrac{\text{arctg(b) + c*b - a/4}}{\text{a*d - 1}}$

31) $$\frac{a + \frac{c}{b} - \sqrt{28*d}}{4*b*a + 1}$$

32) $$\frac{\frac{c}{b} - \sqrt{24+d} + a}{2*a*c - 1}$$

33) $$\frac{2*b - \ln(a + b)*c}{c/4 - 1}$$

34) $$\frac{41 - d/4 - 1}{c/tg(b + a) - d}$$

35) $$\frac{a - b*4 - 1}{c/31 + tg(a*b)}$$

36) $$\frac{lg(b/a + 4) + d}{c - b + 1}$$

37) $$\frac{lg(21 - a)*c/4}{1 + c/a + b}$$

38) $$\frac{c - \ln(33 + b)/4}{a - c/b - 1}$$

39) $$\frac{2*b - 38*c}{arctg(b + a)/c + 1}$$

40) $$\frac{arctg(c/4 + 28)*d}{a/d - c - 1}$$

41) $$\frac{a*b/4 - 1}{\sqrt{41-d} - b*a + c}$$

42) $$\frac{1 + a - b/2}{\sqrt{24+d-c} + a/b}$$

43) $$\frac{\ln(a*b + 2)*c}{41 - b/c + 1}$$

44) $$\frac{lg(4*b - c)*a}{b + c/d - 1}$$

45) $$\frac{2*c + tg(a - 21)}{c/a + d + 1}$$

46) $$\frac{4/c + tg(3*a)}{c/a - b - 1}$$

47) $$\frac{8*lg(b - 1) - c}{a*2 + b/c}$$

48) $$\frac{4*\ln(a/b) + 1}{c + b - d + a}$$

49) $\dfrac{4*\ln(b)/c + 1}{2*c + a - b}$

50) $\dfrac{arctg(b - c)/b + a/4}{a + b - 1}$

51) $\dfrac{arctg(a - c)*b + 28}{4*b/a + 1}$

52) $\dfrac{c*b - 24 + a}{b - \lg(2*c - 1) + a}$

53) $\dfrac{\sqrt{2*b} - a + b/c}{a - c/4 + 1}$

54) $\dfrac{\sqrt{\dfrac{41*d}{a}} + 1}{a - c/b + d}$

55) $\dfrac{a + tg(b/4 - 1)}{c/3 - a*d}$

56) $\dfrac{b*a + c/2}{*c - tg(b + 1)}$

57) $\dfrac{\lg(25 + 2*a/d)}{c + a - b - 1}$

58) $\dfrac{c + 23 - b*4}{a - \ln(a + c/b - 1)}$

59) $\dfrac{b/2 - 53/c}{arctg(b - a)*c + 1}$

60) $\dfrac{c*4 + 28*d/c}{5 - arctg(a*d - c - 1)}$

Контрольные вопросы

1. Особенности выполнения изучаемых базовых команд сопроцессора ix87.
2. Трансцендентные команды.
3. Соглашения о передаче параметров, принятые для используемого в лабораторной работе алгоритмического языка.
4. Состояние стека при выполнении процедур и функций с параметрами.
5. Ассемблирование и дизассемблирование команд на своих примерах.
6. Понятие о байтах кода операции, способах адресации, регистрах и смещениях.
7. Особенности ассемблирования команд сопроцессора.

Пример решения типового варианта

Варианты № 65 и № 66.

Вычислить заданное смешанное арифметическое выражение:

65) $$\frac{\text{arctg}(c/4) - d*62}{a*a + 1}$$

66) $$\frac{-2*c - \ln(d) + 53}{\frac{a}{4} - 1}$$

для данных в форматах **float** (**SINGLE** — переменная **a**) и **int** (**INTEGER** — остальные переменные), используя арифметические операции сопроцессора **ix87**.

В программе на C++ (ПАСКАЛЬ) должны быть предусмотрены функции ввода-вывода. Исходные данные должны вводиться корректно и с проверкой на область допустимых значений. Тип результата — вещественный и определяется из контекста задачи.

Решение

Проанализируем особенности, присущие данным вариантам смешанных арифметических выражений. Сделаем этот пример в реализации Паскаль+Ассемблер. Оптимизацию кода ассемблерных модулей сознательно по-прежнему делать НЕ будем. Если у вас появится такое желание, то это ОЧЕНЬ хорошо. Значит, вы уже ПОНИМАЕТЕ суть дела. Тестовые примеры есть, значит, и ориентир у вас есть. Дерзайте!

У вашего варианта, вполне вероятно, будут другие, свои, СПЕЦИФИЧЕСКИЕ особенности, отличные от тех, которые мы рассматриваем. **Учитесь эти особенности ВИДЕТЬ**. Для этого, как уже указывалось, нужен опыт, а для приобретения опыта надо САМОМУ решать задачи. И, желательно, правильно. Другого пути нет. Итак, в добрый и плодотворный путь!

При выполнении данной лабораторной работы нам понадобятся в основном материалы главы 13. Проиллюстрируем наши действия по шагам.

Реализация в Borland Pascal 7.x и Turbo Assembler 3.2

ШАГ 0.
Анализ особенностей арифметического выражения варианта 65

1. Переменные **a**, **c**, **d** для этого варианта назовем соответственно **a5**, **c5**, **d5**.

2. Результат вычисления нашего смешанного арифметического выражения запомним в переменной **y** типа **Single**. Его диапазона нам хватит, т.к. область определения данных в числителе следующая: **arctg(c/4)=[-p/2,...,p/2]**, **d=[-32768,...,32767]**, а минимальное значение знаменателя равно 1.

3. Внешние данные, передаваемые в Ассемблер, сделаем глобальными.

4. Для команды **FPATAN**, которая нам понадобится для вычисления трансцендентной функции **arctg(c/4)**, существует соотношение **0<c<4** — см. табл. 13.9. Но для процессоров, начиная с **i80386** (согласно документации), таких ограничений на диапазон аргументов уже НЕТ. На всякий случай проверим, так ли это. И если в результате тестовых проверок это подтвердится, то соответствующей проверки в Паскале можно НЕ делать.

5. Можем порадоваться — знаменатель НИКОГДА не будет равен нулю.

ШАГ 1.

Исходный текст модуля Var5.asm

Сделаем максимально простую реализацию, используя явные операнды в командах сопроцессора, и прокомментируем содержимое регистров данных сопроцессора.

```
; вариант 65:   (arctg(c/4) — d*62) / (a*a + 1)
; CopyRight by Голубь Н.Г., 1993
        title var5
        .8087
data        segment      para
        EXTRN a5:Dword,c5:word,d5:word,y:Dword
data        ends
code        segment      para
    assume cs:code,ds:data
        public  var5
four        dd     4
s           dd     62
one         dd     1.0
var5        proc FAR
        finit                                   ;инициализация 8087
                        ;————ST(0)—       !——ST(1)— !—ST(2)—— !—ST(3)—!
                        ;NAN              !NAN          !NAN          !NAN
    fld     one         ;1                !NAN          !NAN          !NAN
    fld     a5          ;a5               !1            !NAN          !NAN
    fmul    st(0),st(0) ;a5*a5            !1            !NAN          !NAN
    fadd    st(0),st(1) ;a5*a5 + 1        !1            !NAN          !NAN
    fild    c5          ;c5               !a5*a5 + 1    !1            !NAN
    fild    four        ;4                !c5           !a5*a5 + 1    !1
    fpatan              ;arctg(c5/4)      !a5*a5 + 1    !1            !NAN
    fild    d5          ;d5               !arctg(c5/4)  !a5*a5 + 1    !1
    fild    s           ;62               !d5           !arctg(c5/4)  !a5*a5+1
    fmul    st(0),st(1) ;62*d5            !d5           !arctg(c5/4)  !a5*a5+1
    fsubr   st(0),st(2) ;arctg(c5/4) -62*d5 !d5         !arctg(c5/4)  !a5*a5+1
    fdiv    st(0),st(3) ;(arctg(c5/4) -62*d5)/(a5*a5 + 1)
    fstp    y
    ret
var5    endp
code        ends
        end
```

ШАГ 2.
Программа для проверки работы процедуры var5

Для организации корректного ввода вещественных данных нужно сделать новую процедуру **InputReal** и добавить ее в файл **INPUTNUM.pas** (см. пример 5.4b). Назовем новый файл **InputNR.pas**.

Исходный текст модуля InputNR.pas

```
Unit InputNR;
{CopyRight by Голубь Н.Г., 1998-2001}
Interface {Заголовки процедур и функций}
{Процедуры ввода целочисленных и вещественных данных с проверкой на
                    ДОПУСТИМЫЙ диапазон:
  A                 — результат КОРРЕКТНОГО ввода
  inv               — строка приглашения на ввод переменной
  Min,Max           — ДОПУСТИМЫЙ диапазон
}
Const
  {Символьные константы}
  inv1='Повторите ввод.';
  inv3='выходит за диапазон [';
  inv4='Вводимое значение ';
Procedure InputNumber(Var A:LongInt;  inv:String; Min,Max:LongInt);
Procedure InputReal  (Var A:Extended; inv:String; Min,Max:Extended);

Implementation {Реализация процедур и функций}
{Процедура корректного ввода целочисленных данных}
Procedure InputNumber(Var A:LongInt; inv:String; Min,Max:LongInt);
Label L;
Var aL    : Real;
    s1    : String;
Begin
 L:
 Repeat
   Write('Введите значение ' ,inv,'===>');
 {Контроль ввода НЕ числовых символов}
 {$I-}
   Readln(aL);      {Промежуточный буфер ввода для ЦЕЛОЧИСЛЕННЫХ данных}
 {$I+}
 Until (IOResult=0);
 if (aL>=Min)and(aL<=Max) then  A:=trunc(aL)
  else
    Begin
      s1:=inv4+inv3;
      Writeln(s1,Min,'..',Max,']!!!!');
      Writeln(inv1);
      goto L;
    End
End;

{Процедура корректного ввода вещественных данных}
Procedure InputReal(Var A:Extended; inv:String; Min,Max:Extended);
Label L;
Var    aL : Extended;
       s1 : String;
       f  : Byte;
```

```
Begin
  L:
{ учет знака для ВЕЩЕСТВЕННЫХ данных:
          f=0 — "+",
          f=1 — "-"
}
  f:=0;
  Repeat
    Write('Введите значение ',inv,'===>');
  {Контроль ввода НЕ числовых символов}
  {$I-}
    Readln(aL);    {Промежуточный буфер ввода для ВЕЩЕСТВЕННЫХ данных}
  {$I+}
  Until (IOResult=0);
{ Специфика проверки диапазона для ОТРИЦАТЕЛЬНЫХ ВЕЩЕСТВЕННЫХ данных }
  If aL <0 then
    begin
     aL:=-aL;
     f:=1;
    end;
  If (aL>=Min)and(aL<=Max) then
    begin
  {Восстановление знака для ОТРИЦАТЕЛЬНЫХ ВЕЩЕСТВЕННЫХ данных}
     If (f=1) then A:=-aL
     else A:=aL
    end
   else
     Begin
      s1:=inv4+inv3;
      Writeln(s1,Min,'..',Max,']!!!!');
      Writeln(inv1);
      goto L;
     End
End;

End.
```



ОБРАТИТЕ ВНИМАНИЕ

на специфику проверки диапазона для ОТРИЦАТЕЛЬНЫХ ВЕЩЕСТВЕННЫХ данных.

Исходный текст главной программы Lab8.pas для тестирования варианта 5

```
{ CopyRight by Голубь Н.Г., 1993, 1998, 2001
        Вычислить арифметическое выражение:
  65)  arctg(c/4) — d*62
      ____________________
             a*a + 1
  с применением СОПРОЦЕССОРА
 }
Program lab8;
{Директивы подключения сопроцессора}
{$N+}
{$E+}
{$L var5}
Uses CRT, InputNR;
```

```
Const
  {Символьные константы}
  invA5='A5 (Single)';
  invC5='C5 (Integer)';
  invD5='D5 (Integer)';
  {Допустимый диапазон для вводимых данных}
  IntMin=-32768;
  IntMax= 32767;
  SingleMin=-1.5e-45; {Help}
  SingleMax= 3.4e38;
Label 1;
Var a5,y   : Single;
   c5,d5   : Integer;
   ch      : char;
   t       : Byte;
   a       : Extended;
   c,d     : LongInt;
Procedure Var5; FAR; external;
Procedure Var5and6;
Begin
   y:= (arctan(c5/4.0) — 62.0*d5)/(a5*a5 + 1);
End;
Begin
 ch:='y';
 t:=1;
 ClrScr;
 writeln   ('=============== Вычисляем: ================");
 writeln   (' y:= (arctan(c5/4) — 62*d5)/(a5*a5 + 1)');
 writeln   ('  ТИП ДАННЫХ:');
 writeln   ('a5,y    : Single;');
 writeln   ('c5,d5  : Integer;');
 while     (ch='y') or (ch='Y') do
  begin
    Window(1,10,79,20);
    ClrScr;
    writeln ('————————— test #',t,'——————————');
    InputReal(a,invA5,SingleMin,SingleMax);
    a5:=a;
    InputNumber(c,invC5,IntMin,IntMax);
    c5:=c;
    InputNumber(d,invD5,IntMin,IntMax);
    d5:=d;
    Writeln ('a5=',a5,';  c5=', c5,';  d5=',d5);
    Var5;
    writeln ('АССЕМБЛЕР: y=',y);
    y:=0;
    Var5and6;
    writeln ('ПАСКАЛЬ:  y=',y);
    writeln ('продолжать? (y/n)');
    ch:=ReadKey;
    t:=t+1;
  end
End.
```

ШАГ 3.

Тестирование реализации для варианта 65

```
=============== Вычисляем: ================
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
   ТИП ДАННЫХ:
a5,y   : Single;
c5,d5  : Integer;
———————— test #1——————————
Введите значение A5 (Single)===>243
Введите значение C5 (Integer)===>1
Введите значение D5 (Integer)===>23433
a5= 2.43000000000000E+0002;  c5=1;  d5=23433;
АССЕМБЛЕР:   y=-2.46036529541016E+0001
ПАСКАЛЬ:     y=-2.46036529541016E+0001
продолжать? (y/n)
=============== Вычисляем: ================
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
   ТИП ДАННЫХ:
a5,y   : Single;
c5,d5  : Integer;
———————— test #2——————————
Введите значение A5 (Single)===>768.758
Введите значение C5 (Integer)===>0
Введите значение D5 (Integer)===>7856
a5= 7.68757995605469E+0002;  c5=0;  d5=7856
АССЕМБЛЕР:   y=-8.24163019657135E-0001
ПАСКАЛЬ:     y=-8.24163019657135E-0001
продолжать? (y/n)
=============== Вычисляем: ================
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
   ТИП ДАННЫХ:
a5,y   : Single;
c5,d5  : Integer;
———————— test #3——————————
Введите значение A5 (Single)===>-567e-22
Введите значение C5 (Integer)===>2
Введите значение D5 (Integer)===>22222
a5=-5.66999998162423E-0020;  c5=2;  d5=22222
АССЕМБЛЕР:   y=-1.37776350000000E+0006
ПАСКАЛЬ:     y=-1.37776350000000E+0006
продолжать? (y/n)
=============== Вычисляем: ================
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
   ТИП ДАННЫХ:
a5,y   : Single;
c5,d5  : Integer;
———————— test #4——————————
Введите значение A5 (Single)===>-67587578.475647
Введите значение C5 (Integer)===>4
Введите значение D5 (Integer)===>23767
a5=-6.75875760000000E+0007;  c5=4;  d5=23767
АССЕМБЛЕР:   y=-3.22576021449805E-0010
ПАСКАЛЬ:     y=-3.22576021449805E-0010
продолжать? (y/n)
```

```
=============== Вычисляем: ===============
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
   ТИП ДАННЫХ:
a5,y  : Single;
c5,d5 : Integer;
———————— test #5———————————-
Введите значение A5 (Single)===>423
Введите значение C5 (Integer)===>24333
Введите значение D5 (Integer)===>3333
АССЕМБЛЕР:    y=-1.15488982200623E+0000
ПАСКАЛЬ:      y=-1.15488982200623E+0000
продолжать? (y/n)
=============== Вычисляем: ===============
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
   ТИП ДАННЫХ:
a5,y  : Single;
c5,d5 : Integer;
———————— test #6———————————-
Введите значение A5 (Single)===>-678678.567
Введите значение C5 (Integer)===>-30000
Введите значение D5 (Integer)===>11111
АССЕМБЛЕР:    y=-1.49560730733356E-0006
ПАСКАЛЬ:      y=-1.49560730733356E-0006
продолжать? (y/n)
```

Тесты 5 и 6 показывают, что процессор **AMD Athlon-650** со встроенным сопроцессором, на котором происходило тестирование, нормально реализует операцию **FPATAN**.



ЗАМЕЧАНИЕ

Если у вас получились ДРУГИЕ результаты, то нужно ставить ограничение для параметра **c** (**0<=c<=4**). Функция ввода в этом случае будет иметь следующий вид:
InputNumber(c,invC5,0,4);

ШАГ 4.
Анализ особенностей арифметического выражения варианта 66

1. Переменные **a**, **c**, **d** для этого варианта назовем соответственно **a6**, **c6**, **d6**.
2. Результат вычисления нашего смешанного арифметического выражения запомним в переменной **z** типа **Extended**. Это, как мы знаем, максимально возможный вещественный тип для сопроцессора — см. табл. П4.3 и табл.13.1. Такой выбор типа результата связан с тем, что минимальное значение знаменателя при значении переменной **a6** типа **Single**, близком к 4 (например, при **a6=4.00000001**), может быть равно 0. Соответственно результат получится **z= -INF** или **z= +INF** (см. тестовый пример #2 — файл **Lab8.tst** в прилагаемых к книге материалах). Надо эту ситуацию НЕ допустить, т.е. проверять на входе значение **(a6/4-1)=0**. **Простая проверка a6=4 для ВЕЩЕСТВЕННЫХ данных в общем случае (и в нашем конкретном случае) НИЧЕГО НЕ дает.** Связано это с точностью представления вещественных чисел, которая зависит от количества разрядов, отводимых под мантиссу для каждого типа данных, — см. п. 2.3.2.
3. Внешние данные, передаваемые в Ассемблер, сделаем глобальными.

4. Для вычисления НАТУРАЛЬНОГО логарифма воспользуемся следующей формулой: **ln(d6)=ln2*log $_2$ (d6)** и командами сопроцессора: **FLDLN2** и **FYL2X** (табл. 13.9.)
5. Значение **d6>0**, поскольку логарифм нуля или отрицательных чисел НЕ существует.

ШАГ 5.
Исходный текст модуля Var6.asm

Для этого варианта тоже сделаем максимально простую реализацию, используя явные операнды в командах сопроцессора, и прокомментируем содержимое регистров данных сопроцессора.

```
; вариант 66: (-2*c — ln(d) + 53)/(a/4 — 1)
; CopyRight by Голубь Н.Г., 1993, 2001
        title var6
        .8087
data        segment  para
        EXTRN a6:Dword,c6:word,d6:word,z: TByte
data        ends
code        segment      para
    assume cs:code,ds:data
        public  var6
four        dd      4.0
s           dd      53.
two         dd      -2
one         dd      1.0
var6        proc FAR
            finit                       ;инициализация 8087
                     ;————————ST(0)———————————!———ST(1)—!—ST(2)———!—ST(3)—!
                     ;NAN                      !NAN         !NAN       !NAN
    fld     one      ;1                        !NAN         !NAN       !NAN
    fld     a6       ;a6                       !1           !NAN       !NAN
    fld     four     ;4                        !a6          !1         !NAN
    fdivr   st(0),st(1) ;a6/4                  !a6          !1         !NAN
    fsub    st(0),st(2) ;a6/4 -1               !a6          !1         !NAN
    fldln2           ;ln2                      !a6/4-1      !a6        !1
    fild    d6       ;d6                       !ln2         !a6/4-1    !a6 ...
    fyl2x            ;ln(d6)=ln2*log2(d6)      !a6/4-1      !a6        !1
    fild    c6       ;c6                       !ln(d6)      !a6/4-1    !a6 ...
    fimul   two      ;-2*c6                    !ln(d6)      !a6/4-1    !a6 ...
    fsub    st(0),st(1) ;-2*c6-ln(d6)          !ln(d6)      !a6/4-1    !a6 ...
    fld     s        ;53                       !-2*c6-ln(d6) !ln(d6)   !a6/4-1 ...
    fadd    st(0),st(1) ;53+(-2*c6-ln(d6))     !-2*c6-ln(d6) !ln(d6)   !a6/4-1 ...
    fdiv    st(0),st(3) ;(53+(-2*c6-ln(d6)))/(a6/4-1)
    fstp    z
    ret
var6        endp
code        ends
end
```

ШАГ 6.
Программа для проверки работы процедуры var6

Добавим в программу на Паскале **строки, относящиеся к реализации варианта 66**. Получим вариант программы, который проверяет работу как варианта 65, так и варианта 66. В более сложных случаях, *когда надо проверить что-то одно, обычно ненужное комментируют.* Мы этого делать НЕ будем. Кроме того, мы можем в процессе совместного тестирования вариантов выловить неточности и в варианте 65, ибо могут появиться какие-то идеи, которые можно тут же и проверить.

Исходный текст главной программы Lab8.pas

```
{ CopyRight by Голубь Н.Г., 1993, 1998, 2001
        Вычислить арифметические выражения:

   65) arctg(c/4) — d*62          |   66) -2*c — ln(d) + 53
     ---------------------        |     -------------------
           a*a + 1                |          a/4 — 1

  с применением СОПРОЦЕССОРА
 }
Program lab8;
{$N+}
{$E+}
{$L var5}
{$L var6}
Uses CRT, InputNR;
Const
  {Символьные константы}
  invA5='A5 (Single)';
  invA6='A6 (Single a6<>4)';
  invC5='C5 (Integer)';
  invC6='C6 (Integer)';
  invD5='D5 (Integer)';
  invD6='D6 (Integer — d6>0)';

  {Допустимый диапазон для вводимых данных и результата}
  IntMin=-32768;
  IntMax= 32767;
  SingleMin=-1.5e-45;
  SingleMax= 3.4e38;

Label 1;
Var a5,a6,y      : Single;
    c5,d5,c6,d6  : Integer;
    ch           : char;
    t            : Byte;
    a,z          : Extended;
    c,d          : LongInt;
Procedure Var5; FAR; external;
Procedure Var6; FAR; external;
Procedure Var5and6;
Begin
 y:= (arctan(c5/4.0) — 62.0*d5)/(a5*a5 + 1);
 z:= (-2.0*c6 — LN(1.0*d6) + 53)/(a6/4 — 1)
```

```
End;
Begin
  ch:='y';
  t:=1;
  ClrScr;
  writeln ('=============== Вычисляем: ================');
  writeln (' y:= (arctan(c5/4) — 62*d5)/(a5*a5 + 1)');
  writeln (' z:= (-2*c6 — LN(d6) + 53)/(a6/4 — 1)');
  writeln('  ТИП ДАННЫХ:');
  writeln('a5,a6,y       : Single;   z:Extended;');
  writeln('c5,d5,c6,d6 : Integer;');
  while (ch='y') or (ch='Y') do
   begin
     Window(1,10,79,20);
     ClrScr;
     writeln ('————————— test #',t,'————————————');
    InputReal(a,invA5,SingleMin,SingleMax);
    a5:=a;
    InputNumber(c,invC5,IntMin,IntMax);
    c5:=c;
    InputNumber(d,invD5,IntMin,IntMax);
    d5:=d;
    1:InputReal(a,invA6,SingleMin,SingleMax);
    a6:=a;
    if (a6/4-1)=0 then   {a=4 НЕЛЬЗЯ!!!!!!!!!!!!!!!!!!!!}
       goto 1;
    InputNumber(c,invC6,IntMin,IntMax);
    c6:=c;
    InputNumber(d,invD6,1,IntMax);
    d6:=d;
    Writeln ('a5=',a5,';  c5=', c5,';  d5=',d5);
    Writeln ('a6=',a6,';  c6=', c6,';  d6=',d6);
    Var5;
    Var6;
    writeln ('АССЕМБЛЕР: y=',y,'  z=',z);
    y:=0;z:=0;
    Var5and6;
    writeln ('ПАСКАЛЬ:   y=',y,'  z=',z);
    writeln ('продолжать? (y/n)');
    ch:=ReadKey;
    t:=t+1;
  end
End.
```

ШАГ 6.
Тестирование реализации для вариантов 65 и 66

```
=============== Вычисляем: ================
 y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
 z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
  ТИП ДАННЫХ:
a5,a6,y       : Single;    z:Extended;
c5,d5,c6,d6 : Integer;
———————— test #1——————————
Введите значение A5 (Single)===>423423.243
```

```
Введите значение C5 (Integer)===>4231
Введите значение D5 (Integer)===>-1243
Введите значение A6 (Single a6<>4)===>4.00000001
Введите значение A6 (Single a6<>4)===>4.0000001
Введите значение A6 (Single a6<>4)===>4.000001
Введите значение C6 (Integer)===>11111
Введите значение D6 (Integer - d6>0)===>32219
a5= 4.23423250000000E+0005;  c5=4231;  d5=-1243
a6= 4.00000095367432E+0000;  c6=11111;  d6=32219
АССЕМБЛЕР:   y= 4.29855276706803E-0007  z=-9.30270635585474E+0010
ПАСКАЛЬ:     y= 4.29855276706803E-0007  z=-9.30270635585474E+0010
продолжать? (y/n)
=============== Вычисляем: ================
 y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
 z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
  ТИП ДАННЫХ:
a5,a6,y      : Single;   z:Extended;
c5,d5,c6,d6 : Integer;
———————— test #2——————————
Введите значение A5 (Single)===>-435e333
Вводимое значение выходит за диапазон [-1.50000000000000E-0045..
3.40000000000000E+0038]!!!!
Повторите ввод.
Введите значение A5 (Single)===>234.2e22
Введите значение C5 (Integer)===>3
Введите значение D5 (Integer)===>423
Введите значение A6 (Single a6<>4)===>4
Введите значение A6 (Single a6<>4)===>44
Введите значение C6 (Integer)===>423
Введите значение D6 (Integer - d6>0)===>0
Вводимое значение выходит за диапазон [1..32767]!!!!
Повторите ввод.
Введите значение D6 (Integer - d6>0)===>423
АССЕМБЛЕР:   y=-4.20389539297445E-0045  z=-7.99047393798828E+0001
ПАСКАЛЬ:     y=-4.20389539297445E-0045  z=-7.99047393798828E+0001
продолжать? (y/n)
=============== Вычисляем: ================
 y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
 z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
  ТИП ДАННЫХ:
a5,a6,y      : Single;   z:Extended;
c5,d5,c6,d6 : Integer;
———————— test #3——————————
Введите значение A5 (Single)===>243
Введите значение C5 (Integer)===>1
Введите значение D5 (Integer)===>23433
Введите значение A6 (Single a6<>4)===>-4
Введите значение C6 (Integer)===>32434
Введите значение D6 (Integer - d6>0)===>23444
a5= 2.43000000000000E+0002;  c5=1;  d5=23433;
a6=-4.00000000000000E+0000;  c6=32434;  d6=23444
АССЕМБЛЕР:   y=-2.46036529541016E+0001  z= 3.24125312500000E+0004
ПАСКАЛЬ:     y=-2.46036529541016E+0001  z= 3.24125312500000E+0004
продолжать? (y/n)
```

```
=============== Вычисляем: ================
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
  z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
   ТИП ДАННЫХ:
a5,a6,y       : Single;    z:Extended;
c5,d5,c6,d6 : Integer;
--------- test #4-----------
Введите значение A5 (Single)===>768.758
Введите значение C5 (Integer)===>0
Введите значение D5 (Integer)===>7856
Введите значение A6 (Single a6<>4)===>0
Введите значение C6 (Integer)===>6789
Введите значение D6 (Integer - d6>0)===>1
a5= 7.68757995605469E+0002;  c5=0;  d5=7856
a6= 0.00000000000000E+0000;  c6=6789;  d6=1
АССЕМБЛЕР:   y=-8.24163019657135E-0001  z= 1.35250000000000E+0004
ПАСКАЛЬ:     y=-8.24163019657135E-0001  z= 1.35250000000000E+0004
продолжать? (y/n)
=============== Вычисляем: ================
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
  z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
   ТИП ДАННЫХ:
a5,a6,y       : Single;    z:Extended;
c5,d5,c6,d6 : Integer;
--------- test #5-----------
Введите значение A5 (Single)===>-567e-22
Введите значение C5 (Integer)===>2
Введите значение D5 (Integer)===>22222
Введите значение A6 (Single a6<>4)===>2112.213e22
Введите значение C6 (Integer)===>32423
Введите значение D6 (Integer - d6>0)===>23433
a5=-5.66999998162423E-0020;  c5=2;  d5=22222
a6= 2.11221307069519E+0025;  c6=32423;  d6=23433
АССЕМБЛЕР:   y=-1.37776350000000E+0006  z=-1.22720693392952E-0020
ПАСКАЛЬ:     y=-1.37776350000000E+0006  z=-1.22720693392952E-0020
продолжать? (y/n)
=============== Вычисляем: ================
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
  z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
   ТИП ДАННЫХ:
a5,a6,y       : Single;    z:Extended;
c5,d5,c6,d6 : Integer;
--------- test #6-----------
Введите значение A5 (Single)===>-67587578.475647
Введите значение C5 (Integer)===>4
Введите значение D5 (Integer)===>23767
Введите значение A6 (Single a6<>4)===>-7465746.675
Введите значение C6 (Integer)===>-32768
Введите значение D6 (Integer - d6>0)===>32767
a5=-6.75875760000000E+0007;  c5=4;  d5=23767
a6=-7.46574650000000E+0006;  c6=-32768;  d6=32767
АССЕМБЛЕР:   y=-3.22576021449805E-0010  z=-3.51357050240040E-0002
ПАСКАЛЬ:     y=-3.22576021449805E-0010  z=-3.51357050240040E-0002
продолжать? (y/n)
```

```
=============== Вычисляем: ================
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
  z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
   ТИП ДАННЫХ:
a5,a6,y      : Single;   z:Extended;
c5,d5,c6,d6 : Integer;
-------- test #7---------
Введите значение A5 (Single)===>423
Введите значение C5 (Integer)===>24333
Введите значение D5 (Integer)===>3333
Введите значение A6 (Single a6<>4)===>23
Введите значение C6 (Integer)===>23233
Введите значение D6 (Integer - d6>0)===>11111
АССЕМБЛЕР:   y=-1.15488982200623E+0000  z=-9.77311914062500E+0003
ПАСКАЛЬ:     y=-1.15488982200623E+0000  z=-9.77311914062500E+0003
продолжать? (y/n)
=============== Вычисляем: ================
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
  z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
   ТИП ДАННЫХ:
a5,a6,y      : Single;   z:Extended;
c5,d5,c6,d6 : Integer;
-------- test #8---------
Введите значение A5 (Single)===>-678678.567
Введите значение C5 (Integer)===>-30000
Введите значение D5 (Integer)===>11111
Введите значение A6 (Single a6<>4)===>-1111111.678
Введите значение C6 (Integer)===>22222
Введите значение D6 (Integer - d6>0)===>1
АССЕМБЛЕР:   y=-1.49560730733356E-0006  z= 1.59806951880455E-0001
ПАСКАЛЬ:     y=-1.49560730733356E-0006  z= 1.59806951880455E-0001
продолжать? (y/n)
=============== Вычисляем: ================
  y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
  z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
   ТИП ДАННЫХ:
a5,a6,y      : Single;   z:Extended;
c5,d5,c6,d6 : Integer;
-------- test #9---------
Введите значение A5 (Single)===>22.333e33
Введите значение C5 (Integer)===>32222
Введите значение D5 (Integer)===>32222
Введите значение A6 (Single a6<>4)===>4.00001
Введите значение C6 (Integer)===>22222
Введите значение D6 (Integer - d6>0)===>32222
a5= 2.23330003334861E+0034;  c5=32222;  d5=32222
a6= 4.00001001358032E+0000;  c6=22222;  d6=32222
АССЕМБЛЕР:   y=-0.00000000000000E+0000  z=-1.77364654701974E+0010
ПАСКАЛЬ:     y=-0.00000000000000E+0000  z=-1.77364654701974E+0010
продолжать? (y/n)
```

Результаты тестов #2 и #9 показывают, что на **шаге 0** при анализе задачи по варианту 65 мы НЕ учли, что в знаменателе **(a5*a5 + 1)** при достаточно больших значениях переменной **a5** может получиться БОЛЬШОЕ число, превышающее диапазон **Single**. А результат получится ОЧЕНЬ МАЛЕНЬКИМ и близким к нулю. Поэтому, если нас интересует более точный результат по варианту 65, нужно переменную **y** сделать типа **Extended**.

ШАГ 7. Обеспечение более точного результата по варианту 65

В этом случае нужно просто заменить описания переменной y как в Паскале, так и в Ассемблере. В Паскале замене подлежат три строки:

```
Var a5,a6         : Single;
      a,y,z       : Extended;
writeln("a5,a6    : Single;  y,z:Extended;");
```

В Ассемблере меняется только одна строка:

```
EXTRN a5:Dword,c5:word,d5:word,y:Tbyte
```

ШАГ 8. Тестирование измененной программы

```
=============== Вычисляем: ===============
 y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
 z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
  ТИП ДАННЫХ:
a5,a6         : Single;   y,z:Extended;
c5,d5,c6,d6 : Integer;
-------- test #1---------
Введите значение A5 (Single)===>22e33
Введите значение C5 (Integer)===>11111
Введите значение D5 (Integer)===>22222
Введите значение A6 (Single a6<>4)===>4.00001
Введите значение C6 (Integer)===>-11111
Введите значение D6 (Integer - d6>0)===>22222
a5= 2.19999994146785E+0034;  c5=11111;  d5=22222
a6= 4.00001001358032E+0000;  c6=-11111;  d6=22222
АССЕМБЛЕР:   y=-2.84661674147905E-0063  z= 8.89391823718623E+0009
ПАСКАЛЬ:     y=-2.84661674147905E-0063  z= 8.89391823718623E+0009
продолжать? (y/n)
=============== Вычисляем: ===============
 y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
 z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
  ТИП ДАННЫХ:
a5,a6         : Single;   y,z:Extended;
c5,d5,c6,d6 : Integer;
-------- test #2---------
Введите значение A5 (Single)===>333.33e-333
Введите значение C5 (Integer)===>11
Введите значение D5 (Integer)===>111
Введите значение A6 (Single a6<>4)===>111
Введите значение C6 (Integer)===>111
Введите значение D6 (Integer - d6>0)===>111
a5= 0.00000000000000E+0000;  c5=11;  d5=111
a6= 1.11000000000000E+0002;  c6=111;  d6=111
АССЕМБЛЕР:   y=-6.88077797467679E+0003  z=-6.49381421313317E+0000
ПАСКАЛЬ:     y=-6.88077797467679E+0003  z=-6.49381421313317E+0000
продолжать? (y/n)
=============== Вычисляем: ===============
 y:= (arctan(c5/4) - 62*d5)/(a5*a5 + 1)
 z:= (-2*c6 - LN(d6) + 53)/(a6/4 - 1)
  ТИП ДАННЫХ:
```

```
a5,a6          : Single;    y,z:Extended;
c5,d5,c6,d6 : Integer;
———————— test #3——————————-
Введите значение A5 (Single)===>-222e-333
Введите значение C5 (Integer)===>11111
Введите значение D5 (Integer)===>11111
Введите значение A6 (Single a6<>4)===>222e333
Вводимое значение выходит за диапазон [-1.50000000000000E-0045..
3.40000000000000E+0038]!!!!
Повторите ввод.
Введите значение A6 (Single a6<>4)===>22e33
Введите значение C6 (Integer)===>22222
Введите значение D6 (Integer — d6>0)===>22222
a5=-0.00000000000000E+0000;  c5=11111;  d5=11111
a6= 2.19999994146785E+0034;  c6=22222;  d6=22222
АССЕМБЛЕР:    y=-6.88880429563677E+0005  z=-8.07291091261459E-0030
ПАСКАЛЬ:      y=-6.88880429563677E+0005  z=-8.07291091261459E-0030
продолжать? (y/n)
=============== Вычисляем: ================
 y:= (arctan(c5/4) — 62*d5)/(a5*a5 + 1)
 z:= (-2*c6 — LN(d6) + 53)/(a6/4 — 1)
  ТИП ДАННЫХ:
a5,a6          : Single;    y,z:Extended;
c5,d5,c6,d6 : Integer;
———————— test #4——————————-
Введите значение A5 (Single)===>-33.222e-33
Введите значение C5 (Integer)===>22222
Введите значение D5 (Integer)===>-32222
Введите значение A6 (Single a6<>4)===>4.00011111
Введите значение C6 (Integer)===>22222
Введите значение D6 (Integer — d6>0)===>32222
a5=-3.32220006058280E-0032;  c5=22222;  d5=-32222
a6= 4.00011110305786E+0000;  c6=22222;  d6=32222
АССЕМБЛЕР:    y= 1.99776557061633E+0006  z=-1.59856555739976E+0009
ПАСКАЛЬ:      y= 1.99776557061633E+0006  z=-1.59856555739976E+0009
продолжать? (y/n)
```

Выводы

1. Если вводить число типа **Single** с ОЧЕНЬ маленьким порядком (см. тесты #2 и #3),то оно вполне логично воспринимается как НОЛЬ.
2. При смешанных вычислениях в Ассемблере ОБЯЗАТЕЛЬНО подключается сопроцессор.
3. Вещественный результат может быть разный в зависимости от типа заявленной переменной. Для переменной **z** по результатам тестирования можно было ограничиться и типом **Single**, а для переменной **y** — типом **double**.
4. В программе на Паскале мы принудительно заставили компилятор подключить сопроцессор путем использования вещественного представления констант при арифметических действиях с целочисленными переменными (см. прил. 8):

   y:= (arctan(c5/**4.0**) — **62.0***d5)/(a5*a5 + 1);

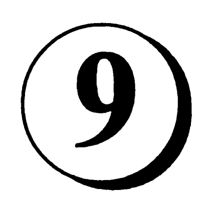
   z:= (-**2.0***c6 — LN(**1.0***d6) + 53)/(a6/4 — 1)

? ВОПРОС

Какой результат на наших (или других) тестах дали бы следующие операторы:

y:= (arctan(c5/**4**) – **62***d5)/(a5*a5 + **1.0**);

z:= (-**2***c6 – LN(d6) + **53.0**)/(a6/**4.0** – 1)



Лабораторная работа № 9

Организация условных переходов, циклов и работа с массивами (сопроцессор ix87)

Квалифицированный специалист — это человек, который удачно избегает маленьких ошибок, неуклонно двигаясь к какому-нибудь глобальному заблуждению.

Следствие Вейнберга
(Артур Блох "Закон Мерфи")

Цель работы

Задав одномерный массив целочисленных или вещественных данных A в одном из заданных форматов (**short int — INTEGER, long int — LONGINT, float — SINGLE, double**), реализовать обработку массива, как указано в варианте. Длина массива N. Исходные данные задать самостоятельно, учитывая формат элементов массива A.

В программе на C++ (ПАСКАЛЬ) должны быть предусмотрены функции ввода-вывода элементов массива A и его обработки. Исходные данные должны вводиться корректно и с проверкой на область допустимых значений. Тип результата должен быть вещественным, а его формат определяется из контекста задачи.

Порядок работы

1) внимательно изучить свой вариант обработки элементов массива;
2) написать на базовом алгоритмическом языке программу ввода исходных данных (с контролем допустимого диапазона), обработки элементов массива и вывода полученного результата;
3) написать модуль обработки элементов массива на языке Ассемблера;

4) встроить вызов этого модуля в программу на базовом алгоритмическом языке;

5) произвести тестовые проверки, отметить нормальные и аномальные результаты, сделать анализ результатов.

Варианты

1 — float (SINGLE); 2 — double; 3 — short int (INTEGER); 4 — long int (LONGINT).
Найти произведение квадратов всех отрицательных элементов массива A={a[i]}.

5 — float (SINGLE); 6 — double; 7 — short int (INTEGER);
8 — long int (LONGINT).
Найти сумму первых K отрицательных элементов массива A={a[i]}.

9 — float (SINGLE); 10 — double; 11 — short int (INTEGER);
12 — long int (LONGINT).
Найти, произведение всех отрицательных элементов массива A={a[i]}, удовлетворяющих условию: c <= a[i] <= d.

13 — float (SINGLE); 14 — double; 15 — short int (INTEGER);
16 — long int (LONGINT).
Найти сумму кубов всех отрицательных элементов массива A={a[i]}.

17 — float (SINGLE); 17 — double; 19 — short int (INTEGER);
20 — long int (LONGINT).
Найти, сумму всех положительных элементов массива A={a[i]}, удовлетворяющих условию: c <= a[i] <= d.

21 — float (SINGLE); 22 — double; 23 — short int (INTEGER);
24 — long int (LONGINT).
Найти сумму квадратов всех положительных элементов массива A={a[i]}.

25 — float (SINGLE); 26 — double; 27 — short int (INTEGER);
28 — long int (LONGINT).
Найти произведение квадратов положительных элементов массива A={a[i]}, удовлетворяющих условию: a[i] >= c.

29 — float (SINGLE); 30 — double; 31 — short int (INTEGER);
32 — long int (LONGINT).
Найти произведение всех элементов массива A={a[i]}, совпадающих с его последним элементом.

33 — float (SINGLE); 34 — double; 35 — short int (INTEGER); 36 — long int (LONGINT).
Найти произведение квадратов отрицательных элементов массива A={a[i]}, удовлетворяющих условию: a[i] <= c.

37 — float (SINGLE); 38 — double; 39 — short int (INTEGER); 40 — long int (LONGINT).
Найти произведение квадратов последних L отрицательных элементов в массиве A={a[i]}.

41 — float (SINGLE); 42 — double; 43 — short int (INTEGER);
44 — long int (LONGINT).

Найти сумму первых K элементов массива A={a[i]}, удовлетворяющих условию: c <= a[i] <= d.

45 — float (SINGLE); 46 — double; 47 — short int (INTEGER);
48 — long int (LONGINT).

Найти сумму всех элементов массива A={a[i]}, совпадающих с его первым элементом.

49 — float (SINGLE); 50 — double; 51 — short int (INTEGER);
52 — long int (LONGINT).

Найти сумму положительных элементов массива A={a[i]}, удовлетворяющих условию: c <= a[i] <= d.

53 — float (SINGLE); 54 — double; 55 — short int (INTEGER);
56 — long int (LONGINT).

Найти сумму последних L положительных элементов в массиве A={a[i]}.

57 — float (SINGLE); 58 — double; 59 — short int (INTEGER);
60 — long int (LONGINT).

Найти произведение всех положительных элементов массива A={a[i]}, удовлетворяющих условию: c <= a[i] <= d.

Контрольные вопросы

1. Особенности выполнения изучаемых базовых команд сопроцессора ix87.
2. Команды безусловного перехода.
3. Команды условного перехода. Организация разветвлений в сопроцессоре.
4. Команды управления циклом.
5. Соглашения о передаче параметров, принятые для используемого в лабораторной работе алгоритмического языка.
6. Состояние стека при выполнении процедур и функций с параметрами.
7. Ассемблирование и дизассемблирование команд на своих примерах.
8. Понятие о байтах кода операции, способах адресации, регистрах и смещениях.
9. Особенности ассемблирования команд сопроцессора.

Пример решения типового варианта

Реализация в Borland C++ 5.02 и Turbo Assembler 4.1

Целочисленный массив

Вариант 62 — short int (INTEGER):

Найти произведение последних L положительных элементов в массиве A={a[i]}. Длина массива N. Исходные данные задать самостоятельно, учитывая формат элементов массива A.

В программе на C++ (ПАСКАЛЬ) должны быть предусмотрены функции ввода-вывода элементов массива A и его обработки. Исходные данные должны вводиться корректно и с проверкой на область допустимых значений. **Тип результата должен быть вещественным, а его формат определяется из контекста задачи.**

Решение

Эту задачу мы с вами уже решали в лабораторных работах № 5 и № 6. Теперь настала пора решить ее с применением команд сопроцессора. Здесь так же, как и в предыдущих лабораторных работах, будут продемонстрированы основные ПРИНЦИПЫ решения ДАННОЙ задачи. ОПТИМИЗАЦИЮ ПРОГРАММЫ, разобравшись с основами, вы сможете сделать и сами.

При выполнении данной лабораторной работы нам понадобятся в основном материалы главы 13 и главы 9. Кроме того, здесь будет полезен и опыт, приобретенный нами при выполнении предыдущей лабораторной работы № 9 и лабораторной работы № 5. Что такое массив мы уже знаем, основные команды — тоже, поэтому дело должно пойти легче и быстрее.

Проиллюстрируем, как обычно, наши действия по шагам.

ШАГ 0.
Анализ особенностей задачи

1. Реализуем в C++ ручной ввод элементов массива, если их количество не превышает 10, в противном случае будем их генерировать. Остановимся пока, как и в лабораторной работе № 5, на максимальном значении количества элементов массива **N=100**.
2. Расширим диапазон случайно генерируемых элементов массива со значения 130 до значения 32767.
3. Сделаем генерацию знака элементов массива теперь тоже случайной, используя функцию **int rand(void)**.
4. В нашей задаче довольно "заковыристый" алгоритм вычисления произведения элементов с КОНЦА массива. Мы с ним достаточно подробно уже разобрались и в языке C++, и в Ассемблере. Оставим его без изменения.
5. Оставим и визуализацию процесса получения произведения как в C++, так и в Ассемблере.

6. Поскольку тип результата должен быть вещественным, а его формат определяется из контекста задачи, то теперь мы сможем существенно расширить диапазон нашего результата — произведения. Примем для результата тип **long double**.
7. Проверять попадание в допустимый диапазон результата в таком случае особого смысла НЕ имеет.
8. Анализ элементов массива в Ассемблере оставим прежним, поскольку НЕТ резона применять достаточно громоздкий механизм реализации разветвлений в сопроцессоре, имея ЦЕЛОЧИСЛЕННЫЕ данные.
9. В качестве прототипа (шаблона) возьмем реализацию нашей задачи в лабораторной работе № 5.

Строки исходного кода, которые были добавлены или изменены по сравнению с прототипом, выделим, как всегда, жирным шрифтом.

ШАГ 1.
ASM-процедура calc

1) Применим команды сопроцессора только при вычислении произведения.
2) Поскольку команды сопроцессора выполняются обычно медленнее, чем команды процессора, то рекомендуется после завершения команды сопроцессора перед началом выполнения команды процессора ставить команду **FWAIT (WAIT)**, иначе можно что-то потерять.
3) Возвращаемое значение, которое прежде было записано через директиву **ARG L:WORD RETURNS res:WORD**, для нашего произведения **res:TBYTE** уже почему-то НЕ срабатывает. Поэтому мы САМИ позаботимся о том, чтобы возвращаемое значение при выходе из процедуры было в вершине стека **ST(0)**— см. п. 9.1.2.2.
4) Добавим команду сохранения и восстановления флагов процессора: **PUSHF** и **POPF**.
5) Чтобы результаты были идентичными как в С++, так и в Ассемблере, сделаем подключение сопроцессора так, как это делает компилятор С++, т.е. через директиву **.286p**.

Исходный текст модуля Func9.asm

```
title Lab_9 (CopyRight by Голубь Н.Г., 1998, 2001)
; Использование СОПРОЦЕССОРА
   .286p
   .MODEL LARGE, C
   LOCALS @@
   .DATA
; ГЛОБАЛЬНЫЕ переменные
EXTRN C array:WORD, array_size:WORD, exist:BYTE

   .CODE
EXTRN    C RangeError: Far ; ВНЕШНЯЯ процедура
   PUBLIC C calc
```

```
; результат вычислений - ТОЛЬКО ТАК!!!
res      DT   ?

arr    EQU  array[BX]
calc   proc far
ARG L:WORD
       mov   exist,0 ; ok!
       FINIT
       FLD1              ;ST(0)=1.0
;выталкивание вершины стека ==> res=1
       FSTP   res
       Fwait
       XOR    SI,SI   ; j=0;
       XOR    BX,BX   ; регистр-индекс
; Подводим указатель BX к последнему элементу
       mov    DI,array_size
       dec    DI           ; array_size-1
       mov    cx, DI
       JCXZ   @@Exit
@@Begin1:
       inc    BX
       inc    BX
       LOOP   @@Begin1
;———————        СОБСТВЕННО РЕШЕНИЕ ЗАДАЧИ:
       mov    cx, array_size
@@Begin:
       mov    DI,arr     ; array[i];
       cmp    DI,0        ;   if (array[i]>0)
       JLE    @@Cont     ; НЕТ
;========= array[i]>0 =========================
       inc    SI        ;   j++;
; сохранение регистров перед вызовом внешней процедуры
       push   ax bx cx dx SI DI
; сохранение флагов перед вызовом внешней процедуры
       pushF
;  Визуализация вычислений на C++
       call   RangeError C, SI,DI
; восстановление флагов
; и регистров после отработки внешней процедуры
       popF
       pop    DI SI dx cx bx ax
;res *= array[i];
       FLD      res                  ;st(0)=res
       FIMUL    arr                  ;st(0)=res*array[i]
       FSTP     res      ;выталкивание вершины стека ==> res
       FWAIT
;if (j==1) return res
       cmp      SI,L
       JE       @@Exit
@@Cont:
;  i--;
```

```
        dec     BX
        dec     BX
        LOOP    @@Begin
        cmp     SI,0      ; нашли положительные элементы?
        JG      @@Exit    ; ДА
        mov     exist,-1  ; НЕТ положительных элементов!!!
@@Exit:
        FLD     res       ; ST(0) — возвращаемое значение!!
        ret
calc    endp
end
```

ШАГ 2.

Фрагменты главной программы на С++

```
/* Lab9.cpp            Borland C++ 5.02
...........................................................
/////////////////////// ВВОД ЭЛЕМЕНТОВ МАССИВА /////////////
//
void inputArray(int A[], int array_size)
{
   randomize(); int j=1;
   for (int i=0; i<array_size; i++)
        if (array_size<11)
     {
          cout << "Enter A[" << i << "]"<< endl;
          InputNum(A[i],-32768L, 32767L);
     }
       else
         {
                j=rand() % 2;//генератор знака числа
                if(!j) j=-1;
                array[i]= j*random (32767);
     }
}
...........................................................
/////////////////////// ОСНОВНАЯ ЗАДАЧА /////////////////////////
//
long double calc_c(const int array[], const int l, const int array_size)
{
  long double res = 1.0;   //!!!!!!!!!!!!!!!!!
  exist=0;
  int i=array_size-1;
  int j=0; // счетчик положительных элементов
  while (i>=0)
   {
    if (array[i]>0)
    {
      j++;
      RangeError(j, array[i]); // Визуализация вычислений
      res = res*array[i];
      exist=1;
      if (j==l) return res;
    }
```

```
    i--;
   }
   return res;
}
///////////////////////////////////////////////////////
// main program
void main()
{
 int l,t=1;
 long double resC,resAsm;
 for(;;)
 {
   textcolor(WHITE);
   clrscr();
   cout << "\n========= test #" << t++ << "===============\n"
        << TITLE << "int array[array_size], 2<=array_size<=" << N << endl;
   cout << ENTER_LEN << "    Ctrl-C (exit)" << endl;
   // Ввод числа элементов массива
   InputNum(array_size);
   // Ввод элементов массива
   inputArray(array, array_size);
   outputArray(array, array_size);
   cout << MULT;
   cout.flush();
   // Ввод числа перемножаемых элементов массива
   InputNum(l, 1, array_size);
   textcolor(WHITE);
   // Вычисления на C++
   cout << "\nC++ function calculation\n";
   resC = 0;
   resC=calc_c(array, l, array_size);
   if (exist == 0) cout << NO_ELEM << endl;
   else     cout << " = " << setprecision(20) << resC << endl;
   // Вычисления на Ассемблере
   cout << "\nAsm function calculation\n";
   resAsm = calc(l);
   if (!exist) cout << " = " << setprecision(20) << resAsm << endl;
   else if (exist == -1) cout << NO_ELEM << endl;
   cout.flush();
   getch();
 }
}
```

ШАГ 3.

Тестирование программы

```
========= test #1===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 22
===================================================================
              Array:
1040   -23859   -28171   -23905     -710    17234    -5248    29529
   13412    -13602 14835    13244   -22190   -26602       519    22694
   -18826   -24055    20808   -20981     -4228    23577
```

```
Введите количество умножаемых элементов
Значение должно быть от 1 до 22 ===> 22
C++ function calculation
23577*20808*22694*519*13244*14835*13412*29529*17234*1040 =
   8.05871368719112861e+39
Asm function calculation
23577*20808*22694*519*13244*14835*13412*29529*17234*1040 =
   8.05871368719112861e+39
========= test #2===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 100
==============================================================
              Array:
21515  17085  31589 -9224  -8079  6148   24482  -3301  -15584 29814
31830  7371   7603  13378  25443  5313   10223  12842  31897  18466
-19763 -8964  1344  8224   -29262 16696  -24467 28477  -9927
18502  -12865 -5329 11349  16447  -22508 -28810 20069  6472   -7852
-9701   -2457 -5917 22412  12216  -838   3210   13833  17322  20723
-26354 32664  -1463 4063   2506   2847   2991   -19578 31260
4217   -12470 -14824 -23047 -18420 14531  14724  13372  -22997 -17442
-12888 6016   -7632 -4852  -14504 -559   10898  -6317  1382
7605   -13626 -24467 -15805 -18965 -16601 15428 -2697  31076  -7672
-20782 32731  -23372      14864  -25110  -21624  -22668  -16145
30611  -26968 -24180  734  -21138

Введите количество умножаемых элементов
Значение должно быть от 1 до 100 ===> 100

C++ function calculation
734*30611*14864*32731*31076*15428*7605*1382*10898*6016*13372*14724*
14531*4217*31260*2991*2847*2506*4063*32664*20723*17322*13833*3210*
12216*22412*6472*20069*16447*11349*18502*28477*16696*8224*1344*18466*
31897*12842*10223*5313*25443*13378*7603*7371*31830*29814*24482*6148*
31589*17085*21515 = 5.15761361051931684e+206

Asm function calculation
734*30611*14864*32731*31076*15428*7605*1382*10898*6016*13372*14724*
14531*4217*31260*2991*2847*2506*4063*32664*20723*17322*13833*3210*
12216*22412*6472*20069*16447*11349*18502*28477*16696*8224*1344*18466*
31897*12842*10223*5313*25443*13378*7603*7371*31830*29814*24482*6148*
31589*17085*21515 = 5.15761361051931684e+206
========= test #3===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 4
Enter A[0]
Значение должно быть от -32768 до 32767 ===> -22
Enter A[1]
Значение должно быть от -32768 до 32767 ===> 22222
Enter A[2]
Значение должно быть от -32768 до 32767 ===> -444
Enter A[3]
Значение должно быть от -32768 до 32767 ===> -2222
==============================================================
```

```
            Array:
       -22    22222     -444    -2222
Введите количество умножаемых элементов
Значение должно быть от 1 до 4 ===> 2

C++ function calculation
22222 = 22222

Asm function calculation
22222 = 22222
========= test #4===============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 5
Enter A[0]
Значение должно быть от -32768 до 32767 ===> -4325
Enter A[1]
Значение должно быть от -32768 до 32767 ===> -1111
Enter A[2]
Значение должно быть от -32768 до 32767 ===> 0
Enter A[3]
Значение должно быть от -32768 до 32767 ===> -2222
Enter A[4]
Значение должно быть от -32768 до 32767 ===> 0
==============================================================
            Array:
    -4325   -1111         0    -2222         0
Введите количество умножаемых элементов
Значение должно быть от 1 до 5 ===> 3

C++ function calculation
C++: В данном массиве НЕТ положительных элементов!!!

Asm function calculation
ASM: В данном массиве НЕТ положительных элементов!!!
```

Теперь мы, наконец, получили полноценное решение нашей задачи!

ШАГ 4.
Как с задачей справился компилятор Borland C++ 5.02

Можем полюбопытствовать, как с решением задачи справился компилятор Borland C++ 5.02. Для этого надо получить ASM-файл: **bcc -S Lab9.cpp**

Найдем в полученном ASM-файле **Lab9.asm** функцию **calc_c**. Компилятор эту функцию сделал в стиле C++, добавив в имя функции информацию о ее параметрах — см. п. 6.1.2.1. **Выделим жирным шрифтом тот код, где непосредственно вычисляется произведение,** и проанализируем его.

Фрагмент файла Lab9.asm

```
@calc_c$qxpxixixt2     proc    near
   ;
   ; long double calc_c(const int array[], const int l, const int array_size)
   ;
    enter   12,0
    push    si
    push    di
   ;
   ; {
   ;        long double res = 1.0;   //!!!!!!!!!!!!!!!!!
   ;
    fld1
    fstp    tbyte ptr [bp-10]
   ;
   ;    exist=0;
   ;
    fwait
    mov     byte ptr DGROUP:_exist,0
   ;
   ;    int i=array_size-1;
   ;
    mov     ax,word ptr [bp+8]
    dec     ax
    mov     si,ax
   ;
   ;    int j=0; // счетчик положительных элементов
   ;
    xor     di,di
    jmp     short @5@6
@5@2:
   ;
   ;    while (i>=0)
   ;     {
   ;       if (array[i]>0)
   ;
    mov     bx,si
    add     bx,bx
    add     bx,word ptr [bp+4]
    cmp     word ptr [bx],0
    jle     short @5@5
   ;
   ;       {
   ;         j++;
   ;
    inc     di
   ;
   ;         RangeError(j, array[i]); // Визуализация вычислений
   ;
    mov     bx,si
    add     bx,bx
    add     bx,word ptr [bp+4]
    push    word ptr [bx]
    push    di
    call    near ptr _RangeError
```

```
     add    sp,4
   ;
   ;        res = res*array[i];
   ;
     mov    bx,si
     add    bx,bx
     add    bx,word ptr [bp+4]
     mov    ax,word ptr [bx]
     mov    word ptr [bp-12],ax
     fild   word ptr [bp-12]
     fld    tbyte ptr [bp-10]
     fmul
     fstp   tbyte ptr [bp-10]
   ;
   ;                      exist=1;
   ;
     fwait
     mov    byte ptr DGROUP:_exist,1
   ;
   ;        if (j==i) return res;
   ;
     cmp    di,word ptr [bp+6]
     jne    short @5@5
     jmp    short @5@7
@5@5:
   ;
   ;     }
   ;    i--;
   ;
     dec    si
@5@6:
     or     si,si
     jge    short @5@2
@5@7:
   ;
   ;        }
   ;        return res;
   ;
     fld    tbyte ptr [bp-10]
     jmp    short @5@8
@5@8:
   ;
   ; }
   ;
     pop    di
     pop    si
     leave
     ret
@calc_c$qxpxixixt2      endp
```

Как видите, мы с компилятором реализовали код немного по-разному. У компилятора получилось немного длиннее. За счет чего?

ЗАМЕЧАНИЕ

Попробуйте увеличить количество элементов в массиве, ну хотя бы до 200. Что получится? Все ли будет нормально считаться?

Вещественный массив

Вариант 63 — float (SINGLE):

Найти произведение последних **L** положительных элементов в массиве **A={a[i]}**. Длина массива **N**. Исходные данные задать самостоятельно, учитывая формат элементов массива **A**.

В программе на С++ (ПАСКАЛЬ) должны быть предусмотрены функции ввода-вывода элементов массива **A** и его обработки. Исходные данные должны вводиться корректно и с проверкой на область допустимых значений. **Тип результата должен быть вещественным, а его формат определяется из контекста задачи.**

Решение

Как видите, это та же задача, которую мы решали в предыдущем случае. Только **изменен тип элементов массива**. Если вы хорошо поняли решение предыдущей задачи, то решение данной задачи будет аналогично предыдущей, только немного более сложное.

Проиллюстрируем, как обычно, наши действия по шагам, указывая только то, что подлежит изменению по сравнению с предыдущей задачей.

ШАГ 0.
Анализ особенностей задачи

1. Сделаем в С++ немного другой алгоритм формирования вещественных элементов массива через целочисленный датчик случайных чисел, если их количество превышает 10.
2. Сделаем нашу программу на С++ более интеллектуальной по вводу, применив динамическую идентификацию типов данных через **typeid**(*объект*).
3. Для проверки допустимого диапазона значений вещественных данных используем константы, которые хранятся в файле **values.h**.
4. Анализ элементов массива в Ассемблере изменим, применив механизм реализации разветвлений в сопроцессоре — см. п. 13.5.3.
5. В качестве прототипа (шаблона) возьмем предыдущую реализацию нашей задачи в этой же лабораторной работе.

Строки исходного кода, которые были добавлены или изменены по сравнению с прототипом, выделим, как всегда, жирным шрифтом.

ШАГ 1.
ASM-процедура calc

```
title Lab_9_2 (CopyRight by Голубь Н.Г., 1998, 2001)
; Использование СОПРОЦЕССОРА
     .286p
   .MODEL LARGE, C
   LOCALS @@
   .DATA
; ГЛОБАЛЬНЫЕ переменные
EXTRN C array:DWORD, array_size:WORD, exist:BYTE
```

```
   .CODE
EXTRN    C RangeError: Far ; ВНЕШНЯЯ процедура
   PUBLIC C calc
; Слово состояния сопроцессора
status dw   ?
; результат вычислений
res       DT   ?
arr    EQU  array[BX]
calc   proc far
ARG L:WORD
       mov    exist,0 ; ok!
       FINIT
       FLD1             ;ST(0)=1.0
;выталкивание вершины стека ==> res=1
       FSTP    res
       Fwait
       XOR    SI,SI   ; j=0;
       XOR    BX,BX   ; регистр-индекс
; Подводим указатель BX к последнему элементу
       mov    DI,array_size
       dec    DI            ; array_size-1
       mov    cx, DI
       JCXZ   @@Exit
@@Begin1:
       ADD     BX,4
       LOOP   @@Begin1
;————————      СОБСТВЕННО РЕШЕНИЕ ЗАДАЧИ:
       mov    cx, array_size
@@Begin:
       FLD   arr            ; st(0)=array[i];
       FLDZ                 ; st(0)=0;     st(1)=array[i];
; Сравнение 0~array[i] с двойным выталкиванием из стека
       FCOMPP
; Запоминание слова состояния программы
       fstsw   status
       fwait
       mov ah,byte ptr status+1
       sahf                 ;ah->cf,pf,af,zf,sf
       JAE    @@Cont        ; if (0>=array[i])goto @@Cont
;========= array[i]>0 ==========================
       inc    SI       ;  j++;
; сохранение регистров
; и флагов перед вызовом внешней процедуры
       push   ax bx cx dx SI
       pushF
;  Визуализация вычислений на C++
       call   RangeError C, SI,arr
; восстановление флагов
; и регистров после отработки внешней процедуры
       popF
```

```
        pop    SI dx cx bx ax
;res *= array[i];
        FLD     res                     ;st(0)=res
        FMUL    arr                     ;st(0)=res*array[i]
        FSTP    res        ;выталкивание вершины стека ==> res
        FWAIT
;if (j==1) return res
        cmp     SI,L
        JE      @@Exit
@@Cont:
;  i--;
        SUB     BX,4
        LOOP    @@Begin
        cmp     SI,0         ; нашли положительные элементы?
        JG      @@Exit       ; ДА
        mov     exist,-1 ; НЕТ положительных элементов!!!
@@Exit:
        FLD     res          ; ST(0) — возвращаемое значение!!!
        ret
calc      endp
end
```

ШАГ 2.

Исходный текст главной программы Lab9_2.cpp

```
/* Lab9_2.cpp              Borland C++ 5.02
       Проект !!!!!! Application.exe === > DOS (standard) !!!!!!
          (c) Copyright 1998-2001 by Голубь Н.Г.
       Лабораторная работа # 9.
                 Использование СОПРОЦЕССОРА.
  Найти произведение последних L положительных элементов массива
          float array[array_size], 2<=array_size<=100
*/
#include <iomanip.h> // манипулятор setprecision(20)
#include <conio.h> //getch(); clrscr();
#include <fstream.h>
#include <stdlib.h> // void randomize(void); int random (int); int rand(void);
#include <values.h>
#include <typeinfo.h> // Динамическая идентификация типов объектов
#include "convert.h"  // Перекодировка КИРИЛЛИЦЫ Win—>DOS
const char* DATA_ERROR = TXT("Ошибка при вводе данных!!!\n");
const char* REPEAT = TXT("Повторите еще раз, пожалуйста...\n");
const char* VALUE = TXT("Значение должно быть от ");
const char* TO = TXT(" до ");
const char* VALUE_RANGE = TXT("Значение превысило допустимый диапазон\n");
const char* TITLE =
       TXT("Найти произведение последних L положительных элементов массива\n");
const char* ENTER_LEN = TXT("Введите длину массива или ");
const char* MEMORY = TXT("НЕВОЗМОЖНО распределить память для массива!!!\n");
const char* MULT = TXT("Введите количество умножаемых элементов\n");
const char* NO_ELEM = TXT("В данном массиве НЕТ положительных элементов!!!\n");
const int N=100;
```

```
//ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
float array[N];
unsigned int array_size;
char exist=0;

extern "C"
{
    long double far calc(int L);
        void RangeError(int i, float buf);
}
///////// Визуализация вычислений в C++ и на Ассемблере ///////////////
//
void RangeError(int j, float buf)
{
  if (j==1) cout << buf;
    else cout << "*" << buf;
}
////////////////////// КОРРЕКТНЫЙ ВВОД //////////////////////
//
template <class typData>
int input (typData& k)
{
    ifstream my_inp ("CON");
    my_inp >> k;
    switch (my_inp.rdstate())
                { case ios::failbit:
                  case ios::badbit:
                        cout << DATA_ERROR << REPEAT;
                        cout.flush();
                        return 1;
                  default:
                        return 0; // ok!
                }
}
///////////////// ВВОД С КОНТРОЛЕМ ДИАПАЗОНА ////////////////////
//
template <class typData>
int InputNum(typData& z, const long double MIN=2L, const long double MAX=N)
{
   long double temp; //!!!!!!!!!!
   int exit_func=0;
// учет знака для ВЕЩЕСТВЕННЫХ данных:
//          f=0 — "+",
//          f=1 — "-"
   int f=0;
   while (exit_func==0)
   {
      cout << VALUE << MIN << TO << MAX << " ===> ";
      cout.flush();
      while (input(temp));
```

```
// Специфика проверки диапазона для ОТРИЦАТЕЛЬНЫХ ВЕЩЕСТВЕННЫХ
//                                          данных
      if(typeid(typData)==typeid(float)||typeid(typData)==typeid(double))
       {
         if (temp<0)
           {
            temp=-temp;
            f=1;
           }
       }
      if ((temp>=MIN)&&(temp<=MAX)) exit_func = 1;
      else
      {
             textcolor(LIGHTRED);
             clreol();
             cout << VALUE_RANGE;
             textcolor(GREEN);
             clreol();
             cout.flush();
      }
    }
// Восстановление знака для ОТРИЦАТЕЛЬНЫХ ВЕЩЕСТВЕННЫХ данных
    if (f)temp=-temp;
    z=(typData)temp;
    return exit_func;
}
//////////////////////// ВВОД ЭЛЕМЕНТОВ МАССИВА //////////////
//
void inputArray(float A[], int array_size, const float MIN=MINFLOAT,
                                       const float MAX=MAXFLOAT)
{
   randomize();
   for (int i=0; i<array_size; i++)
     if (array_size<11)
       {
           cout << "Enter A[" << i << "]"<< endl;
           InputNum(A[i],-MIN,MAX);
       }
     else
      {
       switch(rand() % 2)//генератор знака числа
         { case 0: array[i]= -random (32767)+(double)(i+1)/random (3000);
                 break;
           case 1: array[i]= random (32767)-(double)(i+1)/random (3000);
          }
       }
}
//////////////////////// ВЫВОД ЭЛЕМЕНТОВ МАССИВА //////////////////
//
void outputArray(const float A[], int n)
{ cout << "\n==============================================================\n"
        << "          Array: \n";
  for (int i=0; i<n; i++)
      cout << setprecision(15) << A[i] << ";  ";
  cout << endl;
 }
```

```
////////////////////// ОСНОВНАЯ ЗАДАЧА //////////////////////////
//
long double calc_c(const float array[], const int l, const int array_size)
{
    long double res = 1.0;   //!!!!!!!!!!!!!!!!
   exist=0;
   int i=array_size-1;
   int j=0; // счетчик положительных элементов
   while (i>=0)
    {
      if (array[i]>0)
      {
         j++;
         RangeError(j, array[i]); // Визуализация вычислений
         res *= array[i];
         exist=1;
         if (j==l) return res;
      }
     i--;
   }
return res;
}
/////////////////////////////////////////////////////////////
// main program
void main()
{
 int l,t=1;
 long double resC,resAsm;
 for(;;)
 {
   textcolor(WHITE);
   clrscr();
   cout << "\n========= test #" << t++ << "===============\n"
          << TITLE << "int array[array_size], 2<=array_size<=" << N << endl;
   cout << ENTER_LEN << "    Ctrl-C (exit)" << endl;
   // Ввод числа элементов массива
   InputNum(array_size);
   // Ввод элементов массива
   inputArray(array, array_size);
   outputArray(array, array_size);
   cout << MULT;
   cout.flush();
   // Ввод числа перемножаемых элементов массива
   InputNum(l, 1, array_size);
   textcolor(WHITE);
   // Вычисления на С++
   cout << "\nC++ function calculation\n";
   resC = 0;
   resC=calc_c(array, l, array_size);
   if (exist == 0) cout << NO_ELEM << endl;
   else cout << " = " << setprecision(20) << resC << endl;
   // Вычисления на Ассемблере
   cout << "\nAsm function calculation\n";
   resAsm = calc(l);
```

```
    if (!exist) cout << " = " << setprecision(20) << resAsm << endl;
    else if (exist == -1) cout << NO_ELEM << endl;
    cout.flush();
    getch();
  }
}
```

ШАГ 3.

Тестирование программы

```
========= test #1==============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 22
==============================================================
            Array:
-8952.9970703125;  -25336.99609375;  26756.99609375;  -24329.998046875;
-18371.99609375;  -12117.9970703125; -12759.9736328125;  2900.99682617188;
10350.9765625;  -31684.99609375;  9203.9931640625;  7662.76904296875;
6361.99462890625;  20135.990234375;  -16233.9931640625;  21868.990234375;
-13750.994140625;  -28420.986328125;  -4519.98828125;  -24715.4609375;
-4402.99072265625;  -22237.984375;
Введите количество умножаемых элементов
Значение должно быть от 1 до 22 ===> 1

C++ function calculation
21868.990234375 = 21868.990234375

Asm function calculation
21868.990234375 = 21868.990234375
========= test #2==============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 3
Enter A[0]
Значение должно быть от -1.17549435082228751e-38 до 3.40282346638528886e+38
===> 423243e44
Значение превысило допустимый диапазон
Значение должно быть от -1.17549435082228751e-38 до 3.40282346638528886e+38
===> -243e444
Значение превысило допустимый диапазон
Значение должно быть от -1.17549435082228751e-38 до 3.40282346638528886e+38
===> -234e33
Enter A[1]
Значение должно быть от -1.17549435082228751e-38 до 3.40282346638528886e+38
===> -423e22
Enter A[2]
Значение должно быть от -1.17549435082228751e-38 до 3.40282346638528886e+38
===> 432e22
==============================================================
```

```
                Array:
-2.34000001652108e+35;   -4.23000011334262e+24;   4.32000004829599e+24;
Введите количество умножаемых элементов
Значение должно быть от 1 до 3 ===>3
C++ function calculation
4.32000004829599e+24 = 4.32000004829599329e+24

Asm function calculation
4.32000004829599329e+24 = 4.32000004829599329e+24
========= test #3==============
Найти произведение последних L положительных элементов массива
int array[array_size], 2<=array_size<=100
Введите длину массива или      Ctrl-C (exit)
Значение должно быть от 2 до 100 ===> 2
Enter A[0]
Значение должно быть от -1.17549e-38 до 3.40282e+38 ===> 0
Enter A[1]
Значение должно быть от -1.17549e-38 до 3.40282e+38 ===> -234243
=============================================================
              Array:
0;  -234243;
Введите количество умножаемых элементов
Значение должно быть от 1 до 2 ===> 2

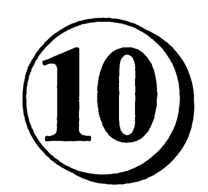
C++ function calculation
В данном массиве НЕТ положительных элементов!!!

Asm function calculation
В данном массиве НЕТ положительных элементов!!!
```

Можно (и нужно) тоже полюбопытствовать, как же эту задачу решает компилятор Borland C++ 5.02. Предоставляю вам сделать этот шаг самостоятельно, уважаемый читатель.

 **ЗАМЕЧАНИЕ**

Константы диапазона для данного типа float, которые хранятся в файле values.h, немного отличаются от того, что рекомендует HELP в Паскале для такого же типа Single, и что написано в прил. 4. Этот разнобой ОЧЕНЬ характерен для сопроцессора. Кроме того, диапазон представления зависит еще и от модели процессора... Где же вы, стандарты?



Лабораторная работа № 10

Организация ввода-вывода целочисленной и текстовой информации

Машина должна работать,
человек – думать.

Принцип IBM (Артур Блох "Закон Мерфи")

Круглое невежество – не самое большое зло: накопление плохо усвоенных знаний еще хуже.

Платон (428 или 427 – 348 или 347 гг. до н.э.)

Цель работы

Реализовать целочисленные вычисления, сделанные в одной из лабораторных работ 3-6 (по указанию преподавателя), полностью используя язык Ассемблера для организации корректного ввода-вывода информации:

- исходные данные должны вводиться с проверкой правильности вводимых символов;
- входные данные и результат должны быть проверены на область допустимых значений;
- при наличии ошибки должно быть выдано соответствующее сообщение.

При реализации задачи в **MS DOS** она должна содержать, как минимум, три макрокоманды, а исполняемый файл должен быть сделан в двух форматах: **EXE**- и **COM**-файлы.

Реализация задачи в операционной системе **Windows** – свободная (для продвинутых студентов).

Порядок работы (реализация в MS DOS)

1) внимательно изучить свой вариант ввода-вывода целочисленной информации применительно к решаемой задаче;
2) написать на языке Ассемблера программу ввода исходных данных (с контролем допустимого диапазона), и вывода полученного результата в одном из предложенных форматов исполняемого файла (EXE- или COM-файл);
3) встроить вызов уже разработанного и отлаженного ранее в лабораторной работе ASM-модуля в программу;
4) произвести тестовые проверки, отметить нормальные и аномальные результаты, сделать анализ результатов;
5) реализовать пункты 2)-4) для другого формата исполняемого файла.

Контрольные вопросы

1. Особенности выполнения изучаемых команд обработки прерываний.
2. Вызов подпрограмм обработки прерываний. Векторы прерываний.
3. Что такое макрокоманды, макроопределения, макрорасширения.
4. Особенности организации EXE-файла.
5. Особенности организации COM-файла.
6. Организация ввода-вывода целочисленной и символьной информации.
7. Ассемблирование и дизассемблирование команд на своих примерах.
8. Необходимость и организация контроля диапазона целочисленных данных при вводе.
9. Понятие о 16- и 32-разрядных вычислительных платформах.
10. Диапазон допустимых значений для целочисленных переменных.

Пример решения типового варианта для MS DOS

Вариант 62 — short int (INTEGER)

Найти произведение последних L положительных элементов в массиве **A={a[i]}**.

Длина массива N. Исходные данные задать самостоятельно, учитывая формат элементов массива **A**.

Решение

Эту задачу мы уже решали – см. лабораторную работу № 5. Но здесь ее решение студент второго курса ХАИ Сергей Левин сделал немного по-другому – см. модуль **array.asm** и тестовые примеры.

Реализуем ввод-вывод данных в MS DOS. При выполнении данной лабораторной работы нам понадобятся в основном материалы глав 14-15. Кроме того, здесь будет полезен и весь предшествующий опыт, приобретенный нами при выполнении лабораторных работ № № 1-6.

Подробно поясним решение нашей задачи для формата EXE-файла. Решение для COM-формата см. в прилагаемых к книге материалах – идейно оно практически ничем НЕ отличается.

Поскольку исходный код достаточно большой, С. Левин решил разбить его на шесть частей (и заодно потренироваться в **создании и использовании модулей**) для формата EXE-файла:

- **array.asm** – главная программа.
- **macros.asm** – макросы.
- **BinAscii.asm** – модуль, содержащий процедуры преобразования чисел в различные системы счисления (Число <- -> Строка).
- **NumInp.asm** – модуль, содержащий процедуры ввода чисел с контролем на допустимый диапазон и некорректные символы.
- **string.asm** – модуль, содержащий процедуры для работы со строками.
- **strio.asm** – модуль, содержащий процедуры для ввода-вывода строк.

Каждый модуль (кроме **macros.asm**) транслируется отдельно (с сохранением полной отладочной информации – ключ **/zi**), а затем собирается редактором связи (компоновщиком):

Пакетный файл make.bat

```
tasm /zi string.asm
tasm /zi strio.asm
tasm /zi binascii.asm
tasm /zi numinp.asm
tasm /zi array.asm
tlink  /v array.obj string.obj strio.obj binascii.obj numinp.obj
```

Программа реализована в стиле TASM (Ideal), снабжена подробными комментариями, поэтому, думаю, проблем с пониманием ее работы у вас, уважаемые мои читатели, не будет. В этой программе есть **интересные моменты**, на которые по ходу описания исходного кода я обращу ваше внимание. Она немного избыточна (профессионал сделал бы, конечно, короче), но зато понятна и сделана с любовью. Здесь **нет той плавающей ошибки**, на которую я намекала при решении примера 14.5, но зато есть другие... Попробуйте их найти и **устранить самостоятельно**!

Реализация в Turbo Assembler 4.1



ОБРАТИТЕ ВНИМАНИЕ

на то, что ВСЕ строки в модуле **array.asm** заканчиваются на символ **'0'**, как стандартные строки в языке С (**строки ASCIZ**), и нигде нет символа **'$'**.

Исходный текст главной программы array.asm

```
; Разработал студент 627 группы ХАИ Левин С.С., 2000 г.
        IDEAL
        MODEL small
        STACK 256
```

```
MaxLen      EQU         128         ; максимальная длина строки
cr          EQU         13          ; возврат каретки
lf          EQU         10          ; перевод строки
MaxSize     EQU         100         ; макс. размер массива — 100
            DATASEG
exCode      DB          0           ; код выхода
Array       DW          MaxSize     DUP(0)
ArraySize   DW          0           ; текущий размер массива
tmp_str     DB          MaxLen DUP(0)
;------------------------------------------------------------------
MSG_ENTER_SIZE         DB    'Введите количество элементов массива', 0
MSG_ENTER_ARRAY_1      DB    cr, lf, 'Введите A[', 0
MSG_ENTER_ARRAY_2      DB    ']', 0
MSG_ENTER_QUANTITY     DB    cr, lf
DB 'Введите количество перемножаемых ПОЛОЖИТЕЛЬНЫХ'
DB 'элементов массива',0
MSG_RANGE_ERROR        DB    cr, lf
db '** Ошибка ** Результат перемножения больше, чем 16 бит', cr, lf
db 'Завершение работы программы',cr, lf, 0
MSG_RESULT             DB    cr, lf, 'Результат перемножения  ', 0
MSG_NOELEMENTS_ERROR DB    cr, lf
Db 'В массиве не обнаружено необходимое количество элементов больше 0', 0
MSG_AGAIN              DB    cr, lf, 'Продолжить? (y/n) ',0
;------------------------------------------------------------------
            CODESEG
;=========== Внешние процедуры:
            EXTRN StrRead:proc, StrWrite:proc, NewLine:proc
            EXTRN SBinToAscDec:proc, DecAscToBin:proc, BinToAscDec:proc
            EXTRN StrLength:proc, ClrScr:proc
            EXTRN EnterUINT:proc, EnterINT:proc
;==============================================
Start:
     mov  ax, @data
     mov  ds, ax
     mov  es, ax
     call ClrScr
     mov  di, offset MSG_ENTER_SIZE
     call StrWrite
     xor  bx, bx              ; установка
     inc  bx                  ;      левого предела
     inc  bx
     mov  dx, MaxSize         ; установка правого предела
     call EnterUINT ; ax <- длина массива
     mov  [ArraySize], ax                ;
     mov  cx, ax              ; установка счетчика цикла
                              ;      для ввода значений массива
     mov  si, offset Array         ; si <- адрес начала массива
@@EnterArray:
     push cx             ; при вызове дальнейших функций
                         ; cx может меняться, но он
                         ; необходим для работы цикла,
                         ; поэтому нужно сохранить его в стеке
     mov  dx, cx              ; сохранить в dx для правильного
                              ; вывода индекса вводимого
                              ; числа в массив
```

```
    mov  di, offset MSG_ENTER_ARRAY_1
    call StrWrite
    mov  di, offset tmp_str  ; di <- адрес строки в которую
                             ; будет осуществляться перевод
                             ; из bin в ASCII текущего индекса
    mov  ax, [ArraySize]     ; получение
    mov  cx, dx              ;      текущего
    sub  ax, cx              ;          индекса
    xor  cx, cx              ; установка
    inc  cx                  ;      минимальной длины строки (1)
    call BinToAscDec         ; преобразование
    call StrWrite            ; вывод индекса
    mov  di, offset MSG_ENTER_ARRAY_2
    call StrWrite
    call EnterINT            ; ax <- введенное значение
    mov  [word si], ax           ; перенос значения в память
    inc  si                  ; сдвиг указателя
    inc  si                  ;      массива на слово
    pop  cx                  ; восстановление итерации цикла
    loop @@EnterArray

    mov  di, offset MSG_ENTER_QUANTITY
    call StrWrite
    xor  bx, bx              ; установка левого
    inc  bx                  ;      предела
    inc  bx
    mov  dx, [ArraySize]     ; установка правого предела
    call EnterUINT             ; ax <- количество перемножаемых
                             ;          элементов массива
    mov  cx, ax              ; инициализация цикла
    mov  bx, [ArraySize]     ; bx <- Размер массива
    dec  bx                  ; индексация начинается с 0
    shl  bx, 1               ; bx <- sizeof(array)
    xor  ax, ax              ; ax <- 0
    inc  ax                  ; ax <- 1
@@10:
    push ax
    push bx
    mov  ax, bx              ; ax <- номер рассматриваемого эл-та
    mov  bx, offset Array    ; ax <- адрес первого эл-та
    add  bx, ax              ; bx <- адрес рассматриваемого эл-та
    mov  dx, [bx]           ; dx <- значение рассматриваемого эл-та
    pop  bx
    pop  ax
    cmp  dx, 0               ; dx <= 0
    jle  @@20                ; если да, то переходим к следующему
                             ; если нет, то
    dec  cx
    push bx
    mov  bx, dx          ; bx <- значение рассматриваемого эл-та
    mul  bx              ; <dx:ax> <- ax * array[i]
    pop  bx
    or   dx, dx
    jnz  @@RangeError
```

```
@@20:
     dec   bx                     ; перемещение
     dec   bx                     ;          указателя на слово назад
     cmp   cx, 0
     je    @@ContinueProcessing
     cmp   bx, 0
     jge   @@10
@@ContinueProcessing:
     or    cx, cx
     jnz   @@NoElements
     mov   di, offset MSG_RESULT      ; вывод
     call  StrWrite                   ;      строки рез-та
     xor   cx, cx                     ; установка минимальной
     inc   cx                         ;     длины строки
     mov   di, offset tmp_str         ; преобразование рез-та
     call  BinToAscDec                ;        в строку
     call  StrWrite                   ; вывод полученной строки
     jmp   Exit
@@NoElements:
     mov   di, offset MSG_NOELEMENTS_ERROR ; вывод сообщения об отсутствии
     call  StrWrite                        ;  положительных элементов
     jmp   Exit
@@RangeError:
     mov   di, offset MSG_RANGE_ERROR      ; вывод сообщения
     call  StrWrite                        ;     об ошибке размера
Exit:
     mov   di, offset MSG_AGAIN
     call  StrWrite
     mov   di, offset tmp_str
     call     StrRead
     cmp   [tmp_str], 'n'
     jne   @@Again
     mov   ah, 04Ch
     mov   al, [exCode]
     int   021h                            ; Выход в DOS
@@Again:
     jmp   Start
END Start
```

Исходный текст файла макросов macros.asm

```
push_r macro reglist
    ifnb <reglist>
      irp reg, <reglist>
           push reg
      endm
    endif
  ifb <reglist>
         push ax bx cx dx si di
  endif
endm
pop_r macro reglist
    ifnb <reglist>
     irp reg, <reglist>
         pop reg
     endm
  endif
```

```
        ifb <reglist>
                pop di si dx cx bx ax
        endif
    endm

    push_r_ax macro reglist
                push bx cx dx si di
    endm

    pop_r_ax macro reglist
                pop di si dx cx bx
    endm

    push_r_cx macro reglist
                push ax bx dx si di
    endm

    pop_r_cx macro reglist
                pop di si dx bx ax
    endm

    ASCNull         EQU         0               ;символ ASCII нуля
```

Модуль **BinAscii.asm** несколько избыточен для решения поставленной задачи, но зато демонстрирует, как можно преобразовывать числа в **разных системах счисления**. Некоторые из этих процедур напоминают известные аналогичные функции языка С (**atoi**, **itoa**).

Исходный текст модуля BinAscii.asm

```
%TITLE "Преобразования Число <- -> Строка  by Левин С., 2000"
include macros.asm
                IDEAL
                MODEL small
                CODESEG
; ————————————— из string.obj ———————————————
                EXTRNStrLength:proc, StrUpper:proc
                PUBLIC    HexDigit, ValCh, NumToAscii
                PUBLIC    SBinToAscDec, BinToAscDec
                PUBLIC    AscToBin, DecAscToBin
;————————————————————————————————————
; HexDigit — преобразует 4-х битовое значение в ASCII-цифру (HEX)
;————————————————————————————————————
; Вход:
;  dl = значение (dl=0..15)
; Выход:
;  dl = HEX эквивалент ASCII цифры
; Регистры:
;  dl
;————————————————————————————————————
PROC HexDigit
    cmp  dl, 10          ; dl<10 ?
    jb   @@10            ; если да, то символ от '0' до '9'
    add  dl, 'A' - 10    ; иначе преобразовать в A, B, C, D, E, F
    ret                  ; возврат в точку вызова
@@10:
    or   dl, '0'   ; если dl<10, то преобразовать в '0'-'9'
```

```
    ret             ; возврат в точку вызова
ENDP HexDigit

;------------------------------------------------
; ValCh — преобразует символ ASCII-цифры в ее двоичное значение
;------------------------------------------------
; Вход:
;   dl = '0'..'9', 'A'..'F'
;   bx = основание (2. 10. 16)
; Выход:
;   cf=0: dx = эквивалентное двоичное значение
;   cf=1: неверный символ для данного основания
; Регистры:
;   dx
;------------------------------------------------
PROC ValCh
        cmp  dl, '9'        ; dl <= '9'
        jbe  @@10
        sub  dl, 7
@@10:
        sub  dl, '0'
        test dl, 0F0h       ; если символ, является числом, то
                            ; единицы могут находиться только
                            ; в младших 4 битах
        jnz  @@99       ; если 1 находятся не только в младших
                        ; разрядах, то  символ — не цифра
        xor  dh, dh         ; dx <- dl
        cmp  dx, bx     ; число-результат должно быть меньше
                            ; основания системы счисления
@@99:
        cmc                 ; инвертировать флаг CF
        ret
ENDP ValCh

;------------------------------------------------
; NumToASCII — преобразует беззнаковое двоичное значение в ASCII
;------------------------------------------------
; Вход:
;   ax = 16 — битовое преобразуемое значение
;   bx = основание результата
;   cx = минимальное количество выводимых цифр
;   di = адрес строки с результатом
; Выход:
;   строка с преобразованным значением
; Регистры:
;   ax, cx
;------------------------------------------------
PROC NumToASCII
        push_r
        xor  si, si     ; обнулить счетчик количества
                        ; преобразованных символов
        jcxz @@20       ; если минимальное количество
                        ; выводимых цифр 0, то переход на 20
```

```
@@10:
        xor  dx, dx      ; <dx:ax> <- ax
        div  bx          ; dx <- остаток от деления <dx:ax>/bx
                         ; ax <- целую часть от
                         ; деления <dx:ax>/bx
        call HexDigit    ; преобразовать значение dl в символ
        push dx          ; сохранить символ в стеке

        inc  si          ; увеличить значение счетчика количества
                         ; преобразованных символов
        loop @@10        ; если cx!=0 тогда: {cx<-cx-1; goto @@10}
@@20:
        inc  cx          ; увеличить cx
        or   ax, ax      ; ax==0 ?
        jnz  @@10        ; если нет, то продолжить преобразования
        mov  cx, si      ; если да, то cx <- количество
                         ; преобразованных символов
; если количество преобразованных символов 0, то goto @@40
        jcxz @@40
        cld              ; DF <- 0 увеличивать значение di
@@30:
        pop  ax          ; восстановить символ из стека
        stosb            ; [di] <- al; di <- di+1
        loop @@30        ; пока (cx != 0) { cx <- cx — 1; goto @@30}
@@40:
        mov  [byte di], ASCNull   ; в конец строки поместить символ
                                  ; конца строки
        pop_r
        ret
ENDP NumToASCII

;------------------------------------------------
; BinToAscDec — преобразует двоичные значения в десятичные ASCII-
  строки
;------------------------------------------------
; Вход:
;   ax = 16-битовое преобразуемое значение
;   cx = минимальное количество выводимых цифр
;   di = адрес строки с результатом
; Выход:
;   строка с преобразованным значением
; Регистры:
;   ax, cx
;------------------------------------------------
PROC BinToAscDec
        push bx             ; сохранить регистр
        mov  bx, 10         ; bx <- основание системы счисления
        call NumToAscii     ; ПРЕОБРАЗОВАТЬ
        pop  bx             ; восстановить регистр

        ret
ENDP BinToAscDec
```

```
;-----------------------------------------
; SBinToAscDec — преобразует знаковые двоичные значения
;               в десятичные ASCII-строки
;-----------------------------------------
; Вход:
;   ax = 16 — битовое преобразуемое значение
;   cx = минимальное количество выводимых цифр
;   di = адрес строки с результатом
; Выход:
;   строка с преобразованным значением
; Регистры:
;   ax, cx
;-----------------------------------------

PROC SBinToAscDec
        push_r
    cmp ax, 0      ; ax == 0
        jge  @@10       ; если ax >= 0 то goto @@10
        neg  ax         ; если нет, то ax <- -ax
        mov  [byte di], '-'  ; в начало строки вставить '-'
        inc  di         ; di <- di+1
@@10:
        mov  bx, 10         ; bx <- основание 10-ой системы
        call NumToAscii ; преобразовать
        pop_r
        ret
ENDP SBinToAscDec

;-----------------------------------------
; ChToBase — возвращает основание системы счисления строки
;-----------------------------------------
; Вход:
;   si = указатель на нулевой ограничитель в конце строки
; Выход:
;   bx = 2, 10, 16
;   si = адрес последней возможной в строке цифры
; Регистры:
;   bx, dl, si
;-----------------------------------------
PROC ChToBase
        mov  dl, [byte si — 1]
        mov  bx, 16
        cmp  dl, 'H'
        je   @@10
        cmp  dl, 'h'
        je   @@10
        mov  bx, 2
        cmp  dl, 'B'
        je   @@10
        cmp  dl, 'b'
        je   @@10
```

```
        mov   bx, 10
        cmp   dl, 'D'
        je    @@10
        cmp   dl, 'd'
        jne   @@20
@@10:
        dec   si
@@20:
        ret
ENDP ChToBase
;-------------------------------------------
; AscToNum — преобразует ASCII-символы цифр в двоичные
;-------------------------------------------
; Вход:
;   ax = начальное значение (0)
;   bx = основание (2, 10, 16)
;   di = адрес строки цифр (БЕЗ знака)
;   si = адрес последней допускаемой в строке цифры
; Выход:
;   cf = 0: ax = беззнаковое значение
;   cf = 1: неверный символ в строке
; Регистры:
;   ax, cx, dx, si
;-------------------------------------------
PROC AscToNum
        xor   cx, cx      ; cx <- 0
        inc   cx          ; cx <- 1; инициализация множителя
@@10:
        cmp   si, di      ; если преобразовывать нечего, то
        je    @@Quit      ; на Выход

        dec   si          ; передвинуть указатель строки
        mov   dl, [byte si]    ; dl <- очередной символ
        call  ValCh; преобразовать символ в цифру
        jc    @@Quit      ; если CF = 1 (неверный символ)
                          ; то на Выход
        push  cx          ; сохранить разряд
        xchg  ax, cx      ; cx<-ax, ax<-cx (подготовка к mul)
        mul   dx          ; ax <- ax * dx
        add   cx, ax      ; к предыдущему рез-ту добавить полученный
        pop   ax          ; восстановить текущий разряд
                          ; (десятки, сотни ...)
        mul   bx          ; увеличить разряд на порядок
        xchg  ax, cx      ; cx <- разряд,
                          ; ax <- результат преобразований
        jmp   @@10
@@Quit:
        ret
ENDP AscToNum
```

```
;---------------------------------------------
; AscToBin - преобразует ASCII-строки в двоичные значения
;---------------------------------------------
; Вход:
;  di = ASCIIZ-строка
; Выход:
;  cf = 0: ax = беззнаковое значение
;  cf = 1: неверный символ в строке
; Регистры:
;  ax
;---------------------------------------------
PROC AscToBin
        push_r_ax
        call StrLength ; cx <-  длина строки
        xor  ax, ax
        jcxz @@Quit    ; если преобразовывать нечего, то Выход

        mov  si, di    ; si <- адрес начала строки
        add  si, cx    ; si передвинуть в конец строки
        cmp  [byte di], '-'  ; если число не отрицательное
        pushf          ; сохранить флаги
        jne  @@10      ; то перейти на метку @@10
        inc  di        ; в противном случае передвинуть
                       ; указатель начала так, чтобы
                       ; был пропущен '-'
@@10:
        call ChToBase  ; bx <- основание системы счисления
        call AscToNum  ; преобразовать строку
        popf           ; восстановить флаги
        jne  @@Quit    ; если '-' нет, то на @@20
        neg  ax        ; в противном случае ax <- -ax
        dec  di        ; передвинуть указатель начала
                       ; строки на символ '-'
@@Quit:
        pop_r_ax
        ret
ENDP AscToBin
;---------------------------------------------
; DecAscToBin - преобразует ASCII-строки (только десятичные числа)
; в двоичные значения
;---------------------------------------------
; Вход:
;  di = ASCIIZ строка
; Выход:
;  cf = 0: ax = беззнаковое значение
;  cf = 1: неверный символ в строке
; Регистры:
;  ax
;---------------------------------------------
```

```
PROC DecAscToBin
        push_r_ax
        call StrLength ; cx <-  длина строки
        xor  ax, ax
        jcxz @@Quit    ; если преобразовывать нечего, то Выход

        mov  si, di         ; si <- адрес начала строки
        add  si, cx         ; si передвинуть в конец строки
        cmp  [byte di], '-' ; если число не отрицательное
        pushf          ; сохранить флаги
        jne  @@10      ; то перейти на метку @@10
        inc  di        ; в противном случае передвинуть
                       ; указатель начала так, чтобы
                       ; был пропущен '-'
@@10:
        mov  bx, 10    ; bx <- основание системы счисления
        call AscToNum  ; преобразовать строку
        rcl  bx, 1     ;
        popf       ; восстановить флаги
        jne  @@20      ; если '-' нет, то на @@20
        neg  ax        ; в противном случае ax <- -ax
        dec  di        ; передвинуть указатель начала
                       ; строки на символ '-'
@@20:
        rcr  bx, 1
@@Quit:
     pop_r_ax
        ret
ENDP DecAscToBin
            END
```

Исходный текст модуля NumInp.asm

```
%TITLE "Ввод чисел с контролем на диапазон и некорректные символы"
include macros.asm
;-------------------------------------------
; CutString убирает ведущие пробелы и лишние нули из строки перед
  обработкой
;-------------------------------------------
CutString macro
     push ax
     mov  al, ' '   ; убрать пробелы
     call CutStr
     mov  al, '0' ; убрать 0
     call CutStr
     pop  ax
endm
               IDEAL
               MODEL small
;------------ Макроопределения ---------------
cr        EQU       13        ; возврат каретки
lf        EQU       10        ; прогон строки
MaxLen    EQU       128
                    DATASEG
```

```
buffer                  DB   128 DUP(0)
MSG_ERROR_SYMBOL DB     cr, lf, lf, '** Ошибка ** Неверный символ в строке '
                        Db   cr, cr, lf, 0
MSG_UINT_RANGE_1        DB   cr,lf,'Число должно быть в диапазоне ', 0
MSG_UINT_RANGE_2        DB   '..', 0
MSG_INT_RANGE           DB   cr,lf
Db 'Число должно быть в диапазоне от минус 32767 до 32767', cr, lf, 0
MSG_RANGE_ERROR  DB cr, lf, lf, '** Ошибка ** Число не входит в диапазон'
                        db cr, cr, lf, 0
                        CODESEG
          EXTRN DecAscToBin:proc, StrRead:proc, StrWrite:proc
          EXTRN NewLine:proc
          EXTRN StrWrite2:proc, StrLength:proc
          EXTRN BinToAscDec:proc
                       PUBLIC EnterUINT, EnterINT

;------------------------------------
;  CutStr - убирает символы, с кодом в AL из начала строки
;------------------------------------
; Вход:
;     al - убираемый символ
;     di - указатель на строку
; Выход:
;     результирующая строка без символов
;------------------------------------
PROC CutStr

     push_r
     cmp  [byte di], '-' ; если число неотрицательное
     jne  @@10           ; то goto @@10
     inc  di             ; в противном случае пропустить '-'
@@10:
     xor  bx, bx         ; количество нулей <- 0
     call StrLength      ; cx <- длина строки
@@Search:
     cmp  [byte di+bx], al     ; под номером bx в строке символ AX
     jne  @@Move    ; если нет, то передвинуть оставшееся в начало
     cmp  bx, cx               ; это конец строки?
     je   @@OnlyOneZero  ; если да, то оставить только один '0'
     inc  bx             ; перейти к следующему символу
     jmp  @@Search       ; проверить снова
@@OnlyOneZero:
     mov  [byte di+1], 0 ; уменьшить длину строки до одного символа
     jmp  @@Quit         ; на Выход
@@Move:
     or   bx, bx         ; нашли ли хоть один 0?
     jz   @@Quit    ; если нет, то менять строку нет необходимости
@@Continue:
     sub  cx, bx         ; cx <- количество перемещаемых символов
     add  bx, di         ; bx <- номер первого не нулевого символа
     mov  si, bx         ; si <- bx
     cld       ; при переносе байтов di и si увеличивать
     rep  movsb          ; перенести байты
     mov  [byte di], 0   ; добавить символ конца строки
```

```
@@Quit:
     pop_r
     ret
ENDP CutStr

;------------------------------------------
;   EnterUINT — производит ввод числа типа unsigned int с клавиатуры
;------------------------------------------
; Вход:
;      dx — правый предел
;      bx — левый предел
; Выход:
;      ax — введенное число
; Регистры
;      ax
;------------------------------------------
PROC EnterUINT
     push_r_ax
@@Again:                                    ; ввести число
     mov  di, offset MSG_UINT_RANGE_1    ;
     call StrWrite                      ; вывести строку диапазона 1
     mov  di, offset buffer             ;
     xor  cx, cx                        ; минимальная длина строки 0
     mov  ax, bx                        ; ax <- левый предел
     call BinToAscDec           ; преобразовать левый предел в строку
     call StrWrite              ; вывести левый предел
     mov  di, offset MSG_UINT_RANGE_2    ;
     call StrWrite                      ; вывести строку диапазона 2
     mov  di, offset buffer             ;
     mov  ax, dx                        ; ax <- правый предел
     xor  cx, cx                        ; cx <- минимальная длина строки
     call BinToAscDec      ; преобразовать правый предел в строку
     call StrWrite         ; вывести правый предел
     call NewLine
     mov  di, offset buffer     ; di <- адрес буфера
     mov  cl, MaxLen            ; cx <- максимальная длина строки
     call StrRead               ; прочитать строку с клавиатуры
     cmp  [byte di], '-'        ; если введен минус,
     je   @@Error               ; то goto @@Error
     CutString
     call StrLength        ; в противном случае cx <- длина строки
     cmp  cx, 5            ; если она больше 5,
     jg   @@Error          ; то goto @@Error
     je   @@Check          ; если символов ровно 5, то необходимо
                           ; проверить на диапазон
@@Continue:                     ; если все в порядке
     call DecAscToBin      ; ax <- число, преобразованное из строки
     jc        @@ErrorSymbol
     cmp  ax, bx           ; проверить на левый предел
     jb   @@Error          ; если меньше, то goto @@Error
     cmp  ax, dx           ; проверить на правый предел
     jg   @@Error          ; если больше, то goto @@Error
     jmp  @@Quit
@@ErrorSymbol:
     mov  di, offset MSG_ERROR_SYMBOL ; в противном случае
     call StrWrite         ; вывести сообщение об ошибке
     jmp  @@Again                ; и повторить ввод
```

```
@@Error:                            ; ошибка диапазонов
      mov   di, offset MSG_RANGE_ERROR; вывести
      call  StrWrite                ; сообщение об ошибке
      jmp   @@Again                 ; и повторить ввод
@@Quit:
      pop_r_ax
      ret   ; возврат в точку вызова
@@Check:
      cmp   [byte di], '6'          ; проверить первый символ
      jg    @@Error                 ; если он больше 6, то goto @@Error
      jb    @@Continue          ; если меньше, то все в порядке
      cmp   [byte di+1], '5'        ; проверить второй символ
      jg    @@Error                 ; если больше 5, то goto @@Error
      jb    @@Continue          ; если меньше, то все в порядке
      cmp   [byte di+2], '5'        ; если равен, то проверить 3-й символ
      jg    @@Error                 ; если больше 5, то goto @@Error
      jb    @@Continue          ; если меньше, то все в порядке
      cmp   [byte di+3], '3'        ; если равен, то проверить 4-й символ
      jg    @@Error                 ; если больше 3, то goto @@Error
      jb    @@Continue          ; если меньше, то все в порядке
      cmp   [byte di+4], '5'        ; если равен, то проверить 5-й символ
      jg    @@Error                 ; если больше 5, то goto @@Error
      jbe   @@Continue          ; если  меньше или равен, то все в порядке
ENDP EnterUINT

;----------------------------------------------
;  EnterINT - производит ввод числа типа int с клавиатуры
;----------------------------------------------
; Вход:
;    нет
; Выход:
;     ax - введенное число
; Регистры
;     ax
;----------------------------------------------
PROC EnterINT
      push_r_ax
@@Again:
      mov   di, offset MSG_INT_RANGE  ;
      call  StrWrite           ; вывести строку диапазона
      mov   di, offset buffer   ;
      mov   cl, MaxLen         ; cl <- максимальная длина строки
      call  StrRead            ; прочитать строку с клавиатуры
      call  StrLength
      cmp   cx, 0
      jz    @@Again
      CutString
      cmp   [byte di], '-' ; проверить на отрицательное число
      jne   @@10               ; если нет, то посчитать длину
                               ; строки на @@10
      inc   di                 ; если да, то пропустить первый символ
      call  StrLength          ; cx <- длина строки
      dec   di                 ; восстановить указатель на '-'
      jmp   @@20               ; пропустить следующую проверку
```

```
@@10:
     call StrLength      ; cx <- длина строки (для полож. числа)
@@20:
     cmp  cx, 5          ;
     jg   @@Error        ; если длина строки больше 5,
                         ; то на @@Error
     je   @@Check        ; если равна, то проверить на диапазон
@@Continue:
     call DecAscToBin    ; преобразовать строку
     jc   @@ErrorSymbol  ; если есть неверные символы,
                         ; то сообщить
     jmp  @@Quit         ; если нет, то на Выход
@@ErrorSymbol:
     mov  di, offset MSG_ERROR_SYMBOL ; вывести

       call    StrWrite        ; сообщение об ошибке (символ)
     jmp  @@Again              ; и повторить ввод
@@Error:
     mov  di, offset MSG_RANGE_ERROR    ; вывести

     call StrWrite             ; сообщение об ошибке (диапазон)
     jmp  @@Again              ; и повторить ввод
@@Quit:
     pop_r_ax
     ret                       ; возврат в точку вызова
@@Check:
     mov  si, di               ; продублировать di
     cmp  [byte di], '-'       ; число отрицательно?
     jne  @@30                 ; если нет, то на @@30
     inc  si                   ; если да, то пропустить '-'
@@30: ; проверить символы на диапазон допустимых значений
     cmp  [byte si], '3'
     jg   @@Error
     jb   @@Continue
     cmp  [byte si+1], '2'
     jg   @@Error
     jb   @@Continue
     cmp  [byte si+2], '7'
     jg   @@Error
     jb   @@Continue
     cmp  [byte si+3], '6'
     jg   @@Error
     jb   @@Continue
     cmp  [byte si+4], '7'
     jg   @@Error
     jbe  @@Continue
ENDP EnterINT
     END
```

Модуль **string.asm** может служить прекрасным **дополнением к примерам главы 11**, в которой мы рассматривали работу с Паскаль-строками. В данном же модуле демонстрируется работа с **С-строками** (строками, которые заканчиваются нулем).

Исходный текст модуля string.asm

```
%TITLE "Процедуры для работы со строками"
include macros.asm
          IDEAL
          MODEL small
          CODESEG
          PUBLIC Move, StrLength, StrCopy

;————————————————————————————————
;    Move —    переместить в памяти блок байтов
;————————————————————————————————
; Вход:
;   si = адрес строки источника
;   di = адрес строки назначения
;   bx = индекс s1(i1)
;   dx = индекс s2(i2)
;   cx = кол-во перемещаемых байтов
; Выход:
;   байты из s1[i1] перемещены в позицию, начинающуюся с s2[i2]
; Регистры:
;   не используются
;————————————————————————————————
PROC Move
         jcxz @@Quit     ;если перемещать нечего, то на Выход
         push_r
; si <- si + номер символа в строке, с которого начать копирование
         add  si, bx
; di <- di + номер символа в строке назначения, куда копировать
         add  di, dx

         cld              ; DF <- 0

         rep  movsb       ; скопировать cx символов из si в di
         pop_r
@@Quit:
         ret
ENDP Move

;————————————————————————————————
; StrLength — подсчитать количество символов в строке
;————————————————————————————————
; Вход:
;   di = адрес строки (s)
; Выход:
;   cx = количество ненулевых символов в строке (s)
; Регистры:
;   не используются
;————————————————————————————————
PROC StrLength
         push_r_cx
         xor  al, al      ; al <- символ конца строки
         mov  cx, 0FFFFh; cx <- -1
         cld              ; DF <- 0 (увеличивать значение di)
repnz    scasb            ; сравнить [di] с al,di <- di+1,cx <- cx-1
         not  cx          ; Логическое отрицание cx
         dec  cx          ; вычитание 1 дает длину строки
         pop_r_cx
         ret
ENDP StrLength
```

```
;-------------------------------------------
; StrCopy - скопировать одну строку в другую
;-------------------------------------------
; Вход:
;   si = адрес строки источника
;   di = адрес строки назначения
; Выход:
;   символы из s1 скопированы в s2
; Регистры:
;   не используются
;-------------------------------------------
PROC StrCopy
        push_r
        xchg si, di         ; di <- адрес строки источника
        call StrLength      ; cx <- длина строки источника
        inc  cx             ; учитывать и символ конца строки
        xchg si, di         ; восстановить значения si и di
        xor  bx, bx         ; копировать с начала
        xor  dx, dx         ;
        call Move           ; скопировать строки
        pop_r
        ret
ENDP StrCopy
END ; конец модуля String
```

ОБРАТИТЕ ВНИМАНИЕ

как реализован вывод строки символов в модуле strio.asm — см. процедуры StrWrite и StrWrite2 (функция DOS 40h — *писать в файл через описатель*, который находится в регистре BX). Поэтому нигде в программе НЕТ символа '$', как это было, например, в примерах главы 14.

Исходный текст модуля strio.asm

```
%TITLE "Процедуры для ввода-вывода строк"
include macros.asm
                    IDEAL
        MODEL small
;------ Макроопределения ----------------
BufSize    EQU  255        ; максимальный размер строки
ASCcr      EQU   13        ; ASCII-символ возврата каретки
ASClf      EQU   10        ; ASCII-символ прогона строки
;---- Структура буфера для DOS функции 0Ah ----------
STRUC StrBuffer
     maxlen    DB   BufSize    ; максимальная длина буфера
     strlen    DB   0          ; длина строки
     charsDB   BufSize DUP (?); буфер для StrRead
ENDS StrBuffer
               DATASEG
buffer         StrBuffer <>    ; переменная буфера для StrRead
               CODESEG
;------------- из string.obj ------------
               EXTRN StrLength:proc, StrCopy:proc
;---------------------------------------
               PUBLIC StrRead, StrWrite, StrWrite2, NewLine
                    PUBLIC ClrScr
;---------------------------------------
```

```
; StrRead — чтение строки с редактированием
;-------------------------------------------
; Вход:
;   di = адрес строки назначения
;   cl = максимальная длина строки
; Выход:
;   Строка, скопированная со стандартного устройства ввода в буфер
; Регистры:
;   не используются
;-------------------------------------------
PROC StrRead
        or   cl, cl          ; cl = 0 ?
        jz   @@Quit          ; если да, то на Выход

        push_r
        mov  [buffer.maxlen], cl   ;установить байт maxlen
        mov  ah, 0Ah               ;функция DOS чтения из буфера
        mov  dx, offset buffer.maxlen ;поместить адрес struct в ds:dx
        int  21h             ;вызов DOS для чтения строки
        xor  bh, bh          ; bh <- 0
        mov  bl, [buffer.strlen] ; bl <- количество введенных символов
        mov  [bx + buffer.chars], ASCNull ; заменить cr на ASCNull
        mov  si, offset buffer.chars     ; si <- адрес буфера
        call StrCopy         ; скопировать буфер в строку
        pop_r
@@Quit:
        ret
ENDP StrRead

;-------------------------------------------
; StrWrite/StrWrite2 — вывести строку на экран (номер устройства
  вывода 1)
;-------------------------------------------
; Вход:
;   di = адрес строки
;   cx = количество записываемых символов (только для StrWrite2)
; Выход:
;   строка s копируется на стандартное устройство вывода
; Регистры:
;   cx
;-------------------------------------------
PROC StrWrite
        call StrLength  ; cx <- длина строки
PROC StrWrite2  ; если в cx уже записана длина строки
        push_r
        xor  bx, bx     ; bx < -0
        inc  bx         ; bx <- 1 (стандартное устройство вывода)
        mov  dx, di     ; строка адресуется в ds:dx
        mov  ah, 040h   ; функция вывода на устройство BX
        int  021h       ; вызов прерывания
        pop_r
        ret
ENDP StrWrite2
ENDP StrWrite
```

```
%NEWPAGE
;-----------------------------------------
; NewLine — перейти на новую строку на стандартном устройстве вывода
;-----------------------------------------
; Вход:
;   нет
; Выход:
;   на стандартное устройство вывода подается символ возврата
;                              каретки и прогона строки
;-----------------------------------------
PROC NewLine
     push ax
     push dx
     mov  ah, 02        ; функция записи символа DOS
     mov  dl, ASCcr     ; dl <- ASCII-символ возврата каретки
     int  021h          ; вывести символ на экран
     mov  dl, ASClf     ; dl <- ASCII-символ прогона строки
     int  021h          ; вывести символ на экран
     pop  dx
     pop  ax
     ret
ENDP NewLine

;-----------------------------------------
; ClrScr — очистка экрана
;-----------------------------------------
; Вход:
;   нет
; Выход:
;   происходит очистка экрана
;-----------------------------------------
PROC ClrScr
     push_r
                        ; очистка экрана
     mov  ah,06h        ; номер функции 10-го прерывания
     mov  al,00h        ; число строк для прокручивания
     mov  ch,00h        ; координата y верхнего левого угла
     mov  cl,00h        ; координата x верхнего левого угла
     mov  dh,024        ; координата y нижнего правого угла
     mov  dl,079        ; координата x нижнего правого угла
     mov  bh,07h        ; атрибуты текста
     int  10h
                        ; перевод курсора в позицию (1,1)
     mov  ah,02h        ; номер функции 10-го прерывания
     mov  bh,00h        ; номер видеостраницы
     mov  dh,0          ; номер столбца
     mov  dl,0          ; номер строки
     int  10h
     pop_r
     ret
ENDP ClrScr
END  ; конец модуля STRIO
```

Тестовые примеры

```
Введите количество элементов массива
Число должно быть в диапазоне 2..100
4
Введите A[0]
Число должно быть в диапазоне от минус 32767 до 32767
-33
Введите A[1]
Число должно быть в диапазоне от минус 32767 до 32767
3
Введите A[2]
Число должно быть в диапазоне от минус 32767 до 32767
333
Введите A[3]
Число должно быть в диапазоне от минус 32767 до 32767
3
Введите количество перемножаемых ПОЛОЖИТЕЛЬНЫХ элементов массива
Число должно быть в диапазоне 2..4
2
Результат перемножения  999
Продолжить (y/n)
Введите количество элементов массива
Число должно быть в диапазоне 2..100
5
Введите A[0]
Число должно быть в диапазоне от минус 32767 до 32767
-2
Введите A[1]
Число должно быть в диапазоне от минус 32767 до 32767
-33
Введите A[2]
Число должно быть в диапазоне от минус 32767 до 32767
-4
Введите A[3]
Число должно быть в диапазоне от минус 32767 до 32767
22
Введите A[4]
Число должно быть в диапазоне от минус 32767 до 32767
22222
Введите количество перемножаемых ПОЛОЖИТЕЛЬНЫХ элементов массива
Число должно быть в диапазоне 2..5
3
** Ошибка ** Результат перемножения больше, чем 16 бит
Завершение работы программы

Продолжить (y/n)
Введите количество элементов массива
Число должно быть в диапазоне 2..100
5
Введите A[0]
Число должно быть в диапазоне от минус 32767 до 32767
-11
Введите A[1]
Число должно быть в диапазоне от минус 32767 до 32767
11
```

```
Введите A[2]
Число должно быть в диапазоне от минус 32767 до 32767
2
Введите A[3]
Число должно быть в диапазоне от минус 32767 до 32767
33
Введите A[4]
Число должно быть в диапазоне от минус 32767 до 32767
-2
Введите количество перемножаемых ПОЛОЖИТЕЛЬНЫХ элементов массива
Число должно быть в диапазоне 2..5
4
В массиве не обнаружено необходимое количество элементов больше 0
Продолжить (y/n)

Введите количество элементов массива
Число должно быть в диапазоне 2..100
4
Введите A[0]
Число должно быть в диапазоне от минус 32767 до 32767
-22
Введите A[1]
Число должно быть в диапазоне от минус 32767 до 32767
22
Введите A[2]
Число должно быть в диапазоне от минус 32767 до 32767
-33
Введите A[3]
Число должно быть в диапазоне от минус 32767 до 32767
33
Введите количество перемножаемых ПОЛОЖИТЕЛЬНЫХ элементов массива
Число должно быть в диапазоне 2..4
2
Результат перемножения  726
Продолжить (y/n)

Введите количество элементов массива
Число должно быть в диапазоне 2..100
e3

** Ошибка ** Неверный символ в строке

Число должно быть в диапазоне 2..100
4
Введите A[0]
Число должно быть в диапазоне от минус 32767 до 32767
-333
Введите A[1]
Число должно быть в диапазоне от минус 32767 до 32767
3
Введите A[2]
Число должно быть в диапазоне от минус 32767 до 32767
333
Введите A[3]
Число должно быть в диапазоне от минус 32767 до 32767
33
Введите количество перемножаемых ПОЛОЖИТЕЛЬНЫХ элементов массива
Число должно быть в диапазоне 2..4
```

```
3
Результат перемножения  32967
Продолжить (y/n)

Введите количество элементов массива
Число должно быть в диапазоне 2..100
6
Введите A[0]
Число должно быть в диапазоне от минус 32767 до 32767
2
Введите A[1]
Число должно быть в диапазоне от минус 32767 до 32767
-3
Введите A[2]
Число должно быть в диапазоне от минус 32767 до 32767
=3
Введите A[3]
Число должно быть в диапазоне от минус 32767 до 32767
333
Введите A[4]
Число должно быть в диапазоне от минус 32767 до 32767
=4
Введите A[5]
Число должно быть в диапазоне от минус 32767 до 32767
44
Введите количество перемножаемых ПОЛОЖИТЕЛЬНЫХ элементов массива
Число должно быть в диапазоне 2..6
2
Результат перемножения  2816
Введите количество элементов массива
Число должно быть в диапазоне 2..100
2
Введите A[0]
Число должно быть в диапазоне от минус 32767 до 32767
3=
Введите A[1]
Число должно быть в диапазоне от минус 32767 до 32767
1
Введите количество перемножаемых ПОЛОЖИТЕЛЬНЫХ элементов массива
Число должно быть в диапазоне 2..2
2
Результат перемножения  36
Продолжить? (y/n)n
```

Как видно из тестовых примеров, НЕТ адекватной реакции на ввод чисел с ошибкой типа =4 или 4=. В чем здесь дело? А может, еще есть где-то, что-то?.. "*Ищите, да обрящете!*"

Пример решения типового варианта. Диалоговое Win32-приложение

Вариант № 61.

Вычислить заданное **условное целочисленное** выражение:

$$X = \begin{cases} 4*a-5, & \text{если } a>2, \\ 22, & \text{если } a=2, \\ (a-9)/2, & \text{если } a<2 \end{cases}$$

для данных в формате **INTEGER (int)**, используя команды сравнения, условного и безусловного переходов. Результат **X** — тоже **целочисленный и его диапазон (формат) зависит от специфики решаемого условного выражения**. Исходные данные должны вводиться корректно (с проверкой на область допустимых значений). Результат также должен быть проверен на область допустимых значений. Задачу реализовать в виде диалогового приложения для **Windows 9x/NT/2000**.

Решение

При выполнении данной лабораторной работы нам понадобятся в основном материалы главы 16 и, конечно, всех предыдущих глав и лабораторных работ. Кроме того, нужна будет библиотека **MSDN**.

Проиллюстрируем наши действия по шагам.

ШАГ 0. Анализ особенностей задачи

Будем вводить целочисленное 16-разрядное данное **a**, результат **X** пусть будет 32-разрядным. Поэтому каких-либо логических или арифметических особенностей у нашей задачи нет.

При программировании на Ассемблере надо будет учесть, что системная библиотека операционных систем **Windows 9x** находится в директории **\system**, а для **Windows NT/2000** — в папке **\system32**. Приложение делается с динамически загружаемыми библиотеками и для успешной его работы в системной директории должна находиться библиотека **shlwapi.dll**.

При Windows-программировании будем использовать библиотеку **SDK** (Win32 API). При обработке строк нужно учесть, что это **С-строки**, т.е. они должны заканчиваться нулем.

Реализация в Turbo Assembler 5.x

ШАГ 1.
Создание главного диалогового окна

Используем любой доступный и удобный редактор ресурсов (можно и несколько редакторов ресурсов). Например, в среде программирования **Borland C++ 5.02** есть удобный для последующего использования в Ассемблере визуальный редактор ресурсов. Он автоматически формирует текстовый **rc**-файл, который можно оттранслировать с помощью программы **brcc32.exe** в двоичный **RES**-файл ресурсов для подключения к EXE-файлу.

Создаем в среде программирования **Borland C++ 5.02** наше главное диалоговое окно (**DIALOG**) и его управляющие элементы (три кнопки – **PUSHBUTTON**, окошко редактирования для ввода числа – **EDITTEXT** и надписи – **LTEXT**, **CTEXT**) – см. рис. Л10.1.

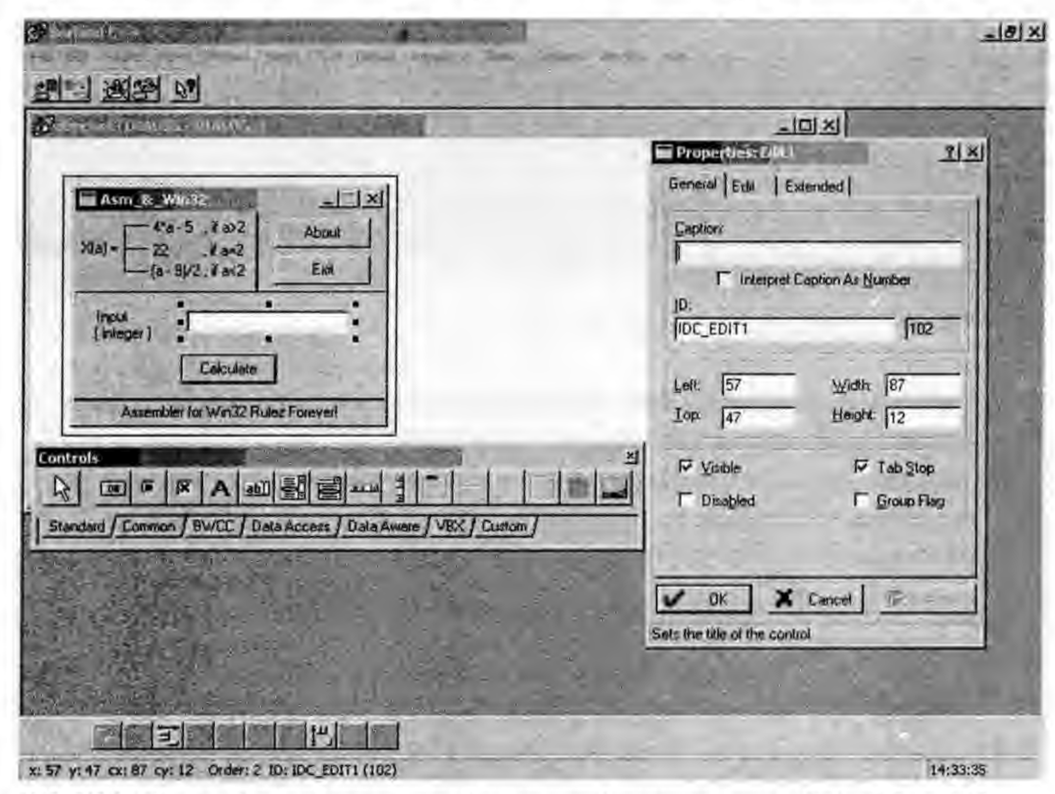


РИС. Л10.1. *Общий вид редактора ресурсов для главного диалогового окна в Borland C++ 5.02.*

За каждым из элементов ресурса закрепляются идентификаторы **IDx** и номера. Так последовательно формируется текстовый **rc**-файл, который имеет и графическую визуализацию – см. рис. Л10.2. Иконку, на мой взгляд, удобнее делать в редакторе ресурсов **Visual C++ 6.0**.

19 Зак. 893

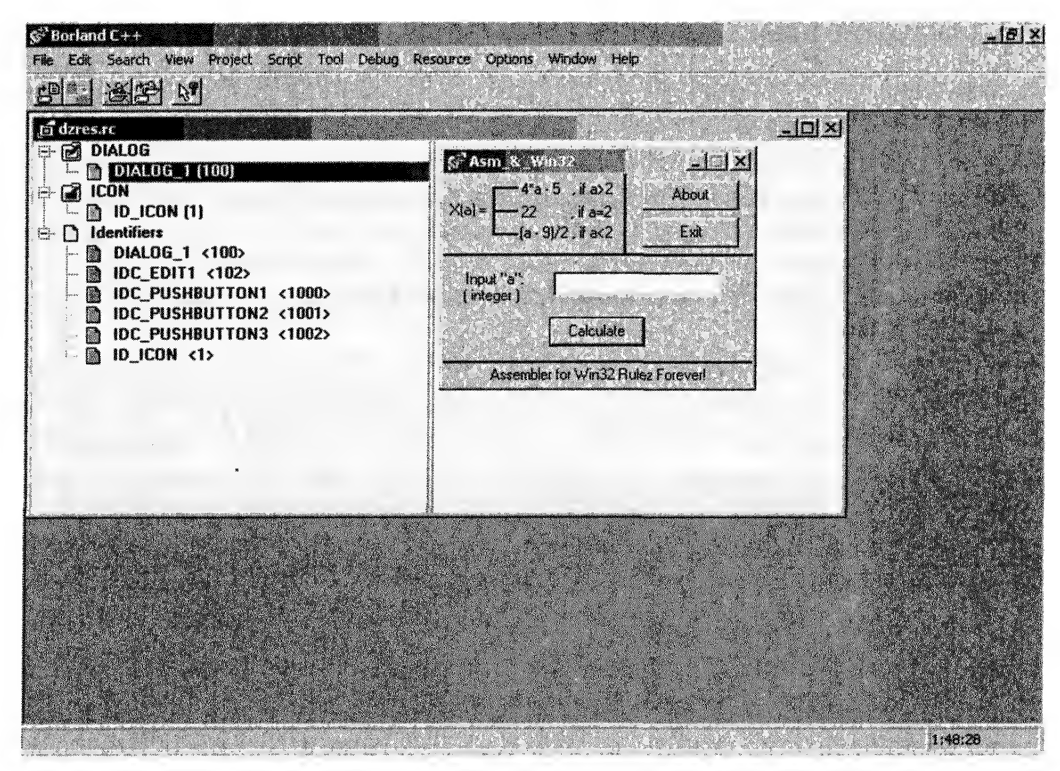


РИС. Л10.2. *Визуализация текстового файла ресурсов dzres.rc.*

В результате получаем файл **dzres.rc**. Он содержит описание всех наших ресурсов:

- номера (дескрипторы) ресурсов;
- размеры и стиль окон и кнопок;
- надписи (если Windows поддерживает русский язык, надписи могут быть и на русском языке);
- название шрифтов и их размер;
- имя файла иконки **dzico.ico**.

Текст файла dzres.rc

```
//
// Igor Patlan 2001// S&I Group
//
#define     ID_ICON              1
ID_ICON  ICON           "dzico.ico"
#define DIALOG_1        100
#define IDC_EDIT1       102
#define IDC_PUSHBUTTON1   1000
#define IDC_PUSHBUTTON2   1001
#define IDC_PUSHBUTTON3   1002
```

```
DIALOG_1 DIALOG DISCARDABLE 32, 42, 161, 102
STYLE DS_3DLOOK | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU |
WS_MINIMIZEBOX
CAPTION "Asm_&_Win32"
FONT 8, "MS Sans Serif"
BEGIN
  DEFPUSHBUTTON "Calculate", IDC_PUSHBUTTON1, 55, 69, 50, 14
  EDITTEXT IDC_EDIT1, 57, 47, 87, 12
  LTEXT "Input ""a"":", -1, 11, 46, 34, 9
  PUSHBUTTON "About", IDC_PUSHBUTTON2, 104, 3, 50, 14
  LTEXT "( integer )", -1, 10, 54, 34, 9
  LTEXT "X(a) =", -1, 3, 13, 22, 10
  LTEXT "4*a — 5   , if a>2", -1, 40, 3, 54, 9, SS_LEFTNOWORDWRAP | WS_GROUP
  LTEXT "22        , if a=2", -1, 40, 14, 53, 9, SS_LEFTNOWORDWRAP | WS_GROUP
  LTEXT "(a — 9)/2 , if a<2", -1, 39, 24, 52, 9, SS_LEFTNOWORDWRAP | WS_GROUP
  CONTROL "", -1, "static", SS_BLACKRECT | WS_CHILD | WS_VISIBLE, 0, 90, 161, 1
  CONTROL "", -1, "static", SS_BLACKRECT | WS_CHILD | WS_VISIBLE, 0, 38, 161, 1
  CONTROL "", -1, "static", SS_BLACKRECT | WS_CHILD | WS_VISIBLE, 25, 6, 1, 23
  CONTROL "", -1, "static", SS_BLACKRECT | WS_CHILD | WS_VISIBLE, 25, 6, 13, 1
  CONTROL "", -1, "static", SS_BLACKRECT | WS_CHILD | WS_VISIBLE, 25, 28, 13, 1
  CONTROL "", -1, "static", SS_BLACKRECT | WS_CHILD | WS_VISIBLE, 25, 17, 13, 1
  CONTROL "", -1, "static", SS_BLACKRECT | WS_CHILD | WS_VISIBLE, 94, 0, 1, 39
  PUSHBUTTON "Exit", IDC_PUSHBUTTON3, 104, 21, 50, 14
  CTEXT "Assembler for Win32 Rulez Forever!", -1, 5, 92, 152, 9
END
```

ШАГ 2.
Изучение функций для работы с DLL

Для создания приложения с динамически подключаемыми библиотеками (**dll**) – см. рис. 16.3 нам необходимы будут соответствующие функции. Обратимся за помощью к MSDN.

- **LoadLibraryA** (***lpLibFileName***) – загружает в память DLL, имя которой указано в переменной ***lpLibFileName*** и которая является **ASCIIZ** строкой. Имя библиотеки должно быть указано полностью, включая полный путь. Если указано только имя, тогда Windows будет искать библиотеку в текущем каталоге. В случае успешного выполнения операции в регистр **EAX** возвращается идентификатор (дескриптор) загруженной в память библиотеки, в противном случае – нуль. Если загружается библиотека, идентификатор которой больше чем 0x7FFF, функция тоже возвратит нуль. **LoadLibraryA** является синонимом функции SDK **LoadLibrary.**
- **GetProcAddress** (***hModule, lpProcName***) – возвращает в регистр **EAX** адрес необходимой процедуры в DLL, имя которой указано в переменной ***lpProcName***, которая является **ASCIIZ**-строкой, а дескриптор библиотеки — в ***hModule.*** Ошибки, произошедшие при выполнении этой процедуры можно определить с помощью процедуры **GetLastError.**
- **FreeLibrary** (***hLibModule***) – выгружает библиотеку из памяти, дескриптор которой содержится в ***hLibModule***. Ошибки, произошедшие при выполнении этой процедуры можно определить с помощью процедуры **GetLastError.**

ШАГ 3.
Изучение функций для работы с диалогами

- **DialogBoxParamA** (***hInstance, lpTemplateName, hWndParent, lpDialogFunc, dwInitParam***) – создание модальной диалоговой формы (окна) из диалогового ресурса. **DialogBoxParamA** является синонимом функции SDK **DialogBoxParam**. Параметры:
  - ***hInstance*** – дескриптор выполняемой задачи (модуля), которая содержит данное диалоговое окно;
  - ***lpTemplateName*** – идентификатор ресурса диалоговой структуры;
  - ***hWndParent*** – дескриптор родительского окна. Если таковых нет, то должен стоять 0, что определяет родителя как *desktop* (рабочий стол) системы.
  - ***lpDialogFunc*** – адрес диалоговой процедуры;
  - ***dwInitParam*** – начальное значение, необходимое как *lParam* в сообщении **WM_INITDIALOG**.
- **GetDlgItemTextA** (***hDlg, nIDDlgItem, lpString, nMaxCount***) – получение строки из диалогового ресурса.
- **GetDlgItemTextA** является синонимом функции SDK **GetDlgItemText**. В случае успешной работы функция возвращает фактическое число символов в строке. Ее параметры:
  - ***hDlg*** – дескриптор (handle) диалогового окна;
  - ***nIDDlgItem*** – идентификатор ресурса, связанного с этим окном;
  - ***lpString*** – адрес **ASCIIZ** строки — текстовый буфер;
  - ***nMaxCount*** – максимальная длина строки.
- **MessageBoxA** (***hWnd, lpText, lpCaption, uType***) – создание модального диалогового окна. **MessageBoxA** (как мы уже знаем – см. п. 16.6.4) является синонимом функции SDK **MessageBox.** Описание параметров:
  - ***hWnd*** – идентификатор родительского окна.
  - ***lpText*** – **ASCIIZ** строка, содержащая текст сообщения.
  - ***lpCaption*** – **ASCIIZ** строка, содержащая заголовок сообщения.
  - ***uType*** – внешний вид (стиль) окна.
- **EndDialog** (***hDlg, nResult***) – функция разрушает диалоговое окно и останавливает все связанные с ним процессы. Параметры:
  - ***hDlg*** – идентификатор разрушаемого диалогового окна.
  - ***nResult*** – значение, которое нужно вернуть приложению, создавшему диалоговое окно.

Шаг 4. Функции получения системной информации

- **GetModuleHandleA** (***lpModuleName***) – возвращает в регистр **EAX** *handle* (идентификатор, дескриптор) модуля (dll- или exe-файла), имя которого содержится в переменной ***lpModuleName***. Если этот параметр имеет значение NULL, то **GetModuleHandleA** возвращает дескриптор процесса, создавшего запрос. Если процедура завершилась неудачно, то в регистр **EAX** возвращается значение 0. Ошибки, произошедшие при выполнении этой процедуры можно определить с помощью процедуры **GetLastError. GetModuleHandleA** является синонимом функции SDK **GetModuleHandle**.
- **GetWindowsDirectoryA** (***lpBuffer, uSize***) – возвращает в символьном массиве ***lpBuffer*** местоположение системной папки Windows, например "C:\WINDOWS". В переменной ***uSize*** указывается максимальный размер этого массива (задается системной константой MAX_PATH). Если функция завершилась успешно, то она возвращает в регистр **EAX** действительное значение длины пути Windows. Если неудачно, то возвращается значение 0. Ошибки, произошедшие при выполнении этой процедуры, можно определить с помощью процедуры **GetLastError. GetWindowsDirectoryA** является синонимом функции SDK **GetWindowsDirectory.**

ШАГ 5. Разработка и описание собственных процедур

- **DlgProc** – основная диалоговая (оконная) процедура. Она является процедурой диалогового окна, которое было создано процедурой ***DialogBoxParamA***. Обрабатывает все сообщения и потоки, поступающие от операционной системы или пользователя в это диалоговое окно, а также выполняет заданные в варианте вычисления.
- **TestProc** – процедура проверяет корректность данных, которые ввел пользователь и, в случае правильного ввода, переводит число из ASCII-символов в его машинное представление, в противном случае устанавливает флаг ошибки в 1.
- **Convert** – процедура переводит число из машинного представления в символьное для вывода на экран.
- **hProc** – фактически это адрес процедуры, находящейся в загружаемой библиотеке. Процедура называется ***StrToIntA***[1] и служит для перевода строки в ее числовое представление.

ШАГ 6. Программирование и отладка приложения (EXE-формат)

Макрос **win32.inc** к данной программе имеет достаточно большой размер, посмотреть его можно в прилагаемых к данной книге материалах. Программа рассчитана на реализацию в **Windows NT/2000** (но оставлен код и для Windows 9x). Автор данной книги намеренно не стала ничего менять в программе, добавив в книжный вариант комментарии, удалив не использующиеся в программе данные и переставив некоторые данные для удобства чтения программы. В прилагаемых к книге материалах оставлен оригинал программы. Вы, уважаемые мои читатели, попробуйте в программе разобраться и сделать ее **более гибкой** к работе в разных операционных системах – это совсем не сложно!.

---

[1] Информацию по этой команде можно найти в справочной системе, поставляемой фирмой Borland со своими компиляторами для программирования под Win32.

В программе использован **расширенный синтаксис команды call** (вызов функций согласуется с моделью **stdcall**) и условные макросы **.if — .endif**. Программа идейно имеет много общего с программой SDKWIN2.C (см. п. 16.6.3). Все строки кода, связанные с **Windows-программированием**, выделены жирным шрифтом.

Исходный текст программы dzasm.asm

```
;
; Разработчик Igor Patlan, 2001 // S&I Group
; Комментарии Голубь Н.Г., 2001
;
;
.386
.model    flat,stdcall
include   win32.inc
includelib     import32.lib
;====== идентификаторы ресурсов:
ID_ICON          equ   1  ; иконка
ID_ARROW               equ   2  ; курсор
IDD_DIALOG             equ   100   ; главное диалоговое окно – см. рис. Л10.1
IDC_EDIT1              equ   102   ; окошко ввода исходных данных
IDC_PUSHBUTTON1  equ   1000  ; кнопка Calculate
IDC_PUSHBUTTON2  equ   1001  ; кнопка About
IDC_PUSHBUTTON3  equ   1002  ; кнопка Exit

.data
;============ Данные:
szCaptn        db    "!!!Win32!!!",0
szZerro        db    "Please, input data!",0
szCapAb        db    "Assembler",0
szCap          db    'Error',0
szTxtAb        db    "   Igor  Patlan",0Ah,0Dh,"     628 group ",0Ah,0Dh
               db    "       2001    ",0
szIncor        db    "You entered incorrect data!",0
szOverf        db    "Calculation excepting overflow!",0
szEq           db    "a=2",0
szGt           db    "a>2",0
szLs           db    "a<2",0
textbuf        db    8 dup (0), 0
Length         dd    ?
ErrFlg         db    0
TmpVar         dd    ? ; значение введенного данного/результата

; дескрипторы:
hProc          dd    0 ; процесса
hDll           dd    0 ; dll
hInst          dd    0 ; окна

; ———— Системная информация ————————
szTxt          db    'Error loading library SHLWAPI.DLL',0
DirLen         dd    0 ; фактическая длина системной директории
; ===== Функция, извлекаемая из dll
DllFun         db    'StrToIntA',0
;===== Windows NT/2000
NTDll          db    '\system32\shlwapi.dll',0
NTLen          dd    $-NTDll-1
DirBuf         db    255 dup (0),0
```

```
;===== Windows 9x
DllNm           db    '\system\shlwapi.dll',0
DllLn           dd    $-DllNm-1
DrBuff          db    255 dup (0),0
;——————————————————————————————————————————

.code
;=========== Программа:
start:
; Запрос длины системной директории Windows и пересылка ее имени в буфер
          call    GetWindowsDirectoryA, offset DirBuf, MAX_PATH
          call    GetWindowsDirectoryA, offset DrBuff, MAX_PATH
   mov    DirLen,eax
   ;for Win32 (Windows 9x)
   lea    edi,DirBuf
   add    edi,DirLen
   lea    esi,DllNm
   mov    ecx,DllLn
   rep    movsb

   ;for WinNT
   lea    edi,DrBuff
   add    edi,DirLen
   lea    esi,NTDll
   mov    ecx,NTLen
   rep    movsb

   ; загрузка системной dll
   call   LoadLibraryA, offset DirBuf
   mov    [hDll],eax
   .if    hDll==0  ; ошибка
          call   LoadLibraryA, offset DrBuff
          mov    [hDll],eax
          .if    hDll==0  ; ошибка
          call   MessageBoxA, 0, offset szTxt, offset szCap, MB_OK or 30h
          call   ExitProcess, 0
          .endif
   .endif

   ; получение адреса дескриптора на нужную dll
   call   GetProcAddress, [hDll], offset DllFun
   mov    hProc, eax

   ; запрос дескриптора процесса – аналог InitApplication (библиотека MFC)
   push   0
   call   GetModuleHandleA
   mov    [hInst],eax
   ; запуск диалога
   call   DialogBoxParamA, hInst, IDD_DIALOG, 0, offset DlgProc, 0
   call   ExitProcess,0   ; окончание работы приложения
;———————————————— конец главной программы ————————————————
;—————————— Оконная процедура вычислений и обработки сообщений ———
DlgProc      proc     hwnd: dword, message: dword, wparam: dword, lparam: dword
; ПЕРЕКЛЮЧАТЕЛЬ switch (message) – обработка сообщений:
   cmp    message,WM_DESTROY          ; разрушить окно?
   je     destroy
   cmp    message,WM_COMMAND          ; выполнить команду?
   je     command
```

```
    xor   eax,eax
    ret

command:   ;======= обработка параметров сообщения WM_COMMAND
    ; ПЕРЕКЛЮЧАТЕЛЬ switch (wparam)
    cmp   wparam, IDC_PUSHBUTTON1
    jz    mess
    cmp   wparam, IDC_PUSHBUTTON2
    jz    about
    cmp   wparam, IDC_PUSHBUTTON3
    jz    destroy
    cmp   wparam, IDCANCEL
    jz    destroy
    xor   eax,eax
    ret

destroy:  ; ——————— разрушить окно
    call  FreeLibrary, [hDll]
    call  EndDialog, hwnd, 0
    ret

mess:  ;——————— выполнить заданные вычисления
    call  GetDlgItemTextA, hwnd, IDC_EDIT1, offset textbuf, 08h
    cmp   eax,6
    ja    err_incor  ; ОШИБКА, если введено более 6 символов
    mov   Length,eax
    xor   eax,eax
    cmp   al,byte ptr textbuf
    jz    err_z
    mov   al,'-'
    cmp   al,byte ptr textbuf
    je    contin
    cmp   Length,5
  ; ОШИБКА, если введено более 5 символов для положительного числа
    ja    err_incor

contin:
    call  TestProc ; проверка корректности введенных данных
    cmp   ErrFlg,1
    jz    err_incor

  ; проверка значения введенного данного
    cmp   TmpVar,2
    jg    vgrate
    jl    vless

    mov   TmpVar,22
    call  Convert
  ; ——————— Вывод результата вычислений, если a = 2
    call  MessageBoxA, hwnd, offset textbuf, offset szEq, MB_OK or 30h
    ret

vgrate:  ; a > 2
    mov   ax,word ptr TmpVar
    cwde
    mov   ecx,4
    imul  ecx
```

```
    sub    eax,5
    mov    TmpVar,eax
    call   Convert
    jmp    _v_gt

vless:  ; a < 2
    mov    ax,word ptr TmpVar
    cwde
    mov    ecx,2
    sub    eax,9
    cdq
    idiv   ecx
    mov    TmpVar,eax
    call   Convert
    jmp    _v_ls

; ============== Выдача сообщений в окошко MessageBox =================
about:  ; ———————— сообщение окна About
    call  MessageBoxA, hwnd, offset szTxtAb, offset szCapAb, MB_OK or 40h
    ret

err_z:  ; ———————— пользователь НЕ ввел данные
    call  MessageBoxA, hwnd, offset szZerro, offset szCaptn, MB_OK or 10h
    ret

err_incor:  ; ———————— Данные НЕ КОРРЕКТНЫ
    call  MessageBoxA, hwnd, offset szIncor, offset szCaptn, MB_OK or 10h
    ret

_v_ls:  ; ———————— Результат вычислений, если a < 2
    call   MessageBoxA, hwnd, offset textbuf,  offset szLs, MB_OK or 30h
    ret

_v_gt:  ; ———————— Результат вычислений, если a > 2
    call   MessageBoxA, hwnd, offset textbuf, offset szGt, MB_OK or 30h
    ret

DlgProc   endp
;———— Конец оконной процедуры ————————————————————
;———————— Анализ корректности данных ————————————————
TestProc  proc
    mov   ErrFlg,0
    mov   esi,offset textbuf
    mov   ecx,Length
    mov   bl,[esi]
    cmp   bl,'-'
    jne   cont
    dec   ecx
    inc   esi
cont:
    mov   bl,[esi]
    sub   bl,30h
    jb    err_dat
    cmp   bl,9
```

```
    ja      err_dat
    inc     esi
    loop    cont
    call    hProc, offset textbuf
    mov     TmpVar,eax
    cmp     eax,00007FFFh
    jg      err_dat
    cmp     eax,0FFFF8000h
    jl      err_dat
    ret
    err_dat:
    mov     ErrFlg,1
    ret

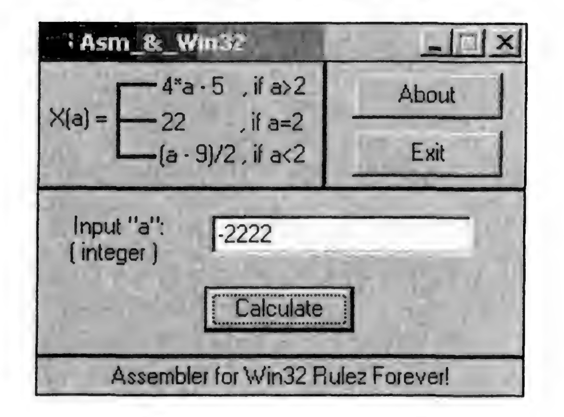
TestProc  endp
;————————— Преобразование число ======> строка —————————————
Convert   proc
    mov     ErrFlg,0
    mov     edi,offset textbuf
    mov     cx,8
    cld
    mov     al,20h
rep         stosb
    dec     edi
    mov     cx,0Ah
    mov     eax,TmpVar
    cmp     eax,0
    jge     next
    neg     eax
    mov     ErrFlg,1
next:
    xor     edx,edx
    cmp     eax,0Ah
    jb      nextend
    div     ecx
    or      dl,30h
    mov     byte ptr [edi],dl
    dec     edi
    jmp     next
nextend:
    or      al,30h
    mov     byte ptr [edi],al
    cmp     ErrFlg,1
    je      WasSg
    ret
WasSg:
    dec     edi
    mov     byte ptr [edi],'-'
    ret
Convert   endp
end       start
```

Теперь мы можем эту программу откомпилировать, запустив наш командный файл с учетом ресурсов (см. п. 16.5):

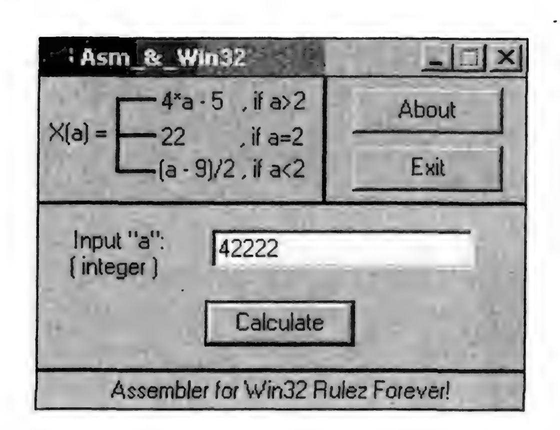
EXE32winRes.bat dzasm.asm dzasm.exe dzres.rc

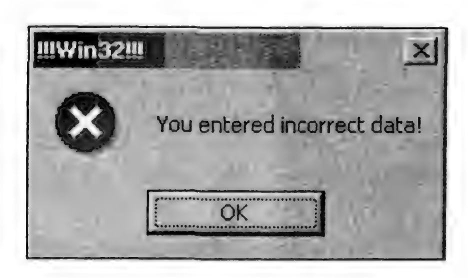
Тестовые примеры

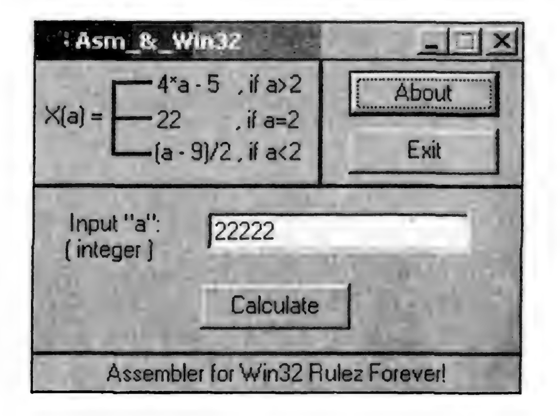
Главное диалоговое окно *Окно результата*





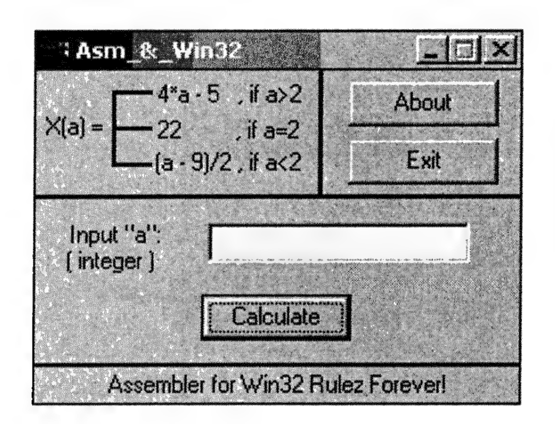


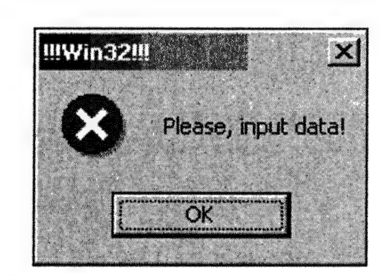


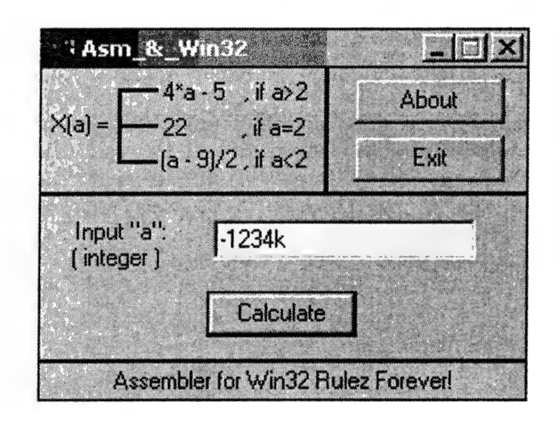


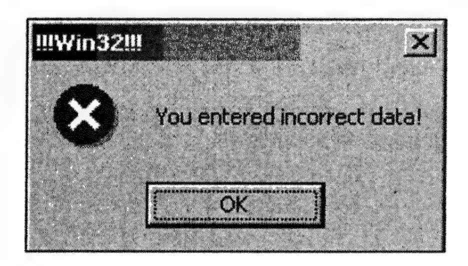


Главное диалоговое окно **Окно результата**









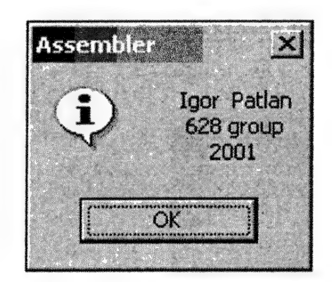


РИС. Л10.3. *Окошко About.*

Вопросы

1. Как реагирует программа **dzasm.exe** на ведущие пробелы?
2. Как отреагируют обе рассмотренные нами в этой лабораторной работе программы на введенное число +55?

Приложения

Приложение 1. Системы счисления

При переводе целых чисел из двоичной системы счисления в десятичную могут понадобиться степени числа 2.

Таблица П1.1. Степени числа 2.

n	2^n	n	2^n
0	1	16	65 536
1	2	17	131 072
2	4	18	262 144
3	8	19	524 288
4	16	20	1 048 576
5	32	21	2 097 152
6	64	22	4 194 304
7	128	23	8 388 608
8	256	24	16 777 216
9	512	25	33 554 432
10	1 024	26	67 108 864
11	2 048	27	134 217 728
12	4 096	28	268 435 456
13	8 192	29	536 870 912
14	16 384	30	1 073 741 824
15	32 768	31	2 147 483 648

Шестнадцатеричное счисление позволяет в удобной и компактной форме представить двоичные числа, с которыми оперирует любая ЭВМ. Одно шестнадцатеричное число — это тетрада (четыре) двоичных.

Таблица П1.2. Связь между десятичной (Decimal), двоичной (Binary) и шестнадцатеричной (Hexadecimal) системами счисления.

Dec	Bin	Hex	Dec	Bin	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Таблица П1.3. Связь между десятичной (Decimal)и шестнадцатеричной (Hexadecimal) системами счисления.

Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec
1	1	10	16	100	256	1000	4096	10000	65536
2	2	20	32	200	512	2000	8192	20000	131072
3	3	30	48	300	768	3000	12288	30000	196608
4	4	40	64	400	1024	4000	16384	40000	262144
5	5	50	80	500	1280	5000	20480	50000	327680
6	6	60	96	600	1536	6000	24576	60000	393216
7	7	70	112	700	1792	7000	28672	70000	458752
8	8	80	128	800	2048	8000	32768	80000	524288
9	9	90	144	900	2304	9000	36864	90000	589824
A	10	A0	160	A00	2560	A000	40960	A0000	655360
B	11	B0	176	B00	2816	B000	45056	B0000	720896
C	12	C0	192	C00	3072	C000	49152	C0000	786432
D	13	D0	208	D00	3328	D000	53248	D0000	851968
E	14	E0	224	E00	3584	E000	57344	E0000	917504
F	15	F0	240	F00	3840	F000	61440	F0000	983040

Продолжение Табл. П1.3.

Hex	Dec	Hex	Dec	Hex	Dec
100000	1048576	1000000	16777216	10000000	268435456
200000	2097152	2000000	33554432	20000000	536870912
300000	3145728	3000000	50331648	30000000	805306368
400000	4194304	4000000	67108864	40000000	1073741824
500000	5242880	5000000	83886080	50000000	1342177280
600000	6291456	6000000	100663296	60000000	1610612736
700000	7340032	70000000	117440512	70000000	1879048192
800000	8388608	8000000	134217728	80000000	2147483648
900000	9437184	9000000	150994944	90000000	2415919104
A00000	10485760	A000000	167772160	A0000000	2684354560
B00000	11534336	B000000	184549376	B0000000	2952790016
C00000	12582912	C000000	201326592	C0000000	3221225472
D00000	13631488	D000000	218103808	D0000000	3489660928
E00000	14680064	E000000	234881024	E0000000	3758096384
F00000	15728640	F000000	251658240	F0000000	4026531840

Приложение 2. Кодировка символов

Американский стандартный код для обмена информацией (American Standard Code for Information Interchange — **ASCII**) — это основной 7-ми битовый двоичный код, в котором представляются алфавитные и цифровые символы, знаки и управляющие команды. В таблице П2.1 показаны **печатаемые (32..126)** и **управляющие (0..31, 127) символы** — **Char** (в десятичной и в шестнадцатеричной системах счисления — соответственно **Dec** и **Hex**). В табл. П2.2 дана расшифровка управляющих символов (0..31).

Таблица П2.1. Таблица стандартных кодов ASCII (0–127).

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0		32	20		64	40	@	96	60	‘
1	1		33	21	!	65	41	A	97	61	a
2	2		34	22	"	66	42	B	98	62	b
3	3		35	23	#	67	43	C	99	63	c
4	4		36	24	$	68	44	D	100	64	d
5	5		37	25	%	69	45	E	101	65	e
6	6		38	26	&	70	46	F	102	66	f
7	7		39	27	'	71	47	G	103	67	g
8	8		40	28	(	72	48	H	104	68	h
9	9		41	29	)	73	49	I	105	69	i
10	A		42	2A	*	74	4A	J	106	6A	j
11	B		43	2B	+	75	4B	K	107	6B	k
12	C		44	2C	,	76	4C	L	108	6C	l
13	D		45	2D	-	77	4D	M	109	6D	m
14	E		46	2E	.	78	4E	N	110	6E	n
15	F		47	2F	/	79	4F	O	111	6F	o
16	10		48	30	0	80	50	P	112	70	p
17	11		49	31	1	81	51	Q	113	71	q
18	12		50	32	2	82	52	R	114	72	r
19	13		51	33	3	83	53	S	115	73	s
20	14		52	34	4	84	54	T	116	74	t
21	15		53	35	5	85	55	U	117	75	u
22	16		54	36	6	86	56	V	118	76	v
23	17		55	37	7	87	57	W	119	77	w
24	18		56	38	8	88	58	X	120	78	x
25	19		57	39	9	89	59	Y	121	79	y
26	1A		58	3A	:	90	5A	Z	122	7A	z
27	1B		59	3B	;	91	5B	[	123	7B	{
28	1C		60	3C	<	92	5C	\	124	7C	|
29	1D		61	3D	=	93	5D	]	125	7D	}
30	1E		62	3E	>	94	5E	^	126	7E	~
31	1F		63	3F	?	95	5F	_	127	7F	·

Таблица П2.2. Управляющие коды ASCII (0-31).

Dec	Hex	Ctrl	Имя	Назначение	Dec	Hex	Ctrl	Имя	Назначение
0	00	^@	NUL	Пусто (конец строки)	16	10	^P	DLE	Оставить канал данных
1	01	^A	SOH	Начало заголовка	17	11	^Q	DC1	Управление устройством-1
2	02	^B	STX	Начало текста	18	12	^R	DC2	Управление устройством-2
3	03	^C	ETX	Конец текста	19	13	^S	DC3	Управление устройством-3
4	04	^D	EOT	Конец передачи	20	14	^T	DC4	Управление устройством-4
5	05	^E	ENQ	Запрос	21	15	^U	NAK	Отрицательное подтверждение
6	06	^F	ACK	Подтверждение	22	16	^V	SYN	Синхронизация
7	07	^G	BEL	Звонок	23	17	^W	ETB	Конец блока передачи
8	08	^H	BS	Шаг назад	24	18	^X	CAN	Отмена
9	09	^I	HT	Горизонтальная табуляция (TAB)	25	19	^Y	EM	Конец носителя
10	0a	^J	LF	Перевод строки	26	1a	^Z	SUB	Подстановка
11	0b	^K	VT	Вертикальная табуляция	27	1b	^[	ESC	Escape
12	0c	^L	FF	Перевод страницы	28	1c	^\	FS	Разделитель файлов
13	0d	^M	CR	Возврат каретки	29	1d	^]	GS	Разделитель групп
14	0e	^N	SO	Выдвинуть	30	1e	^^	RS	Разделитель записей
15	0f	^O	SI	Сдвинуть	31	1f	^_	US	Разделитель полей

Обычно ASCII-кодировка символов одинакова для всех IBM PC совместимых компьютеров. **Остальные коды (128..255)** соответствуют так называемым **национальным алфавитам, символам псевдографики и прочим символам,** которые, естественно, отличаются для разных стран. Отличаются они, к сожалению, и для разных операционных систем. В табл. П2.3 приведены символы, соответствующие **альтернативной кодировке ГОСТа для символов кириллицы** (русские буквы), псевдографики и прочих символов — кодировка **ср-866**. Эта кодировка используется как основная в DOS—приложениях и как транспортная — в компьютерной сети Fidonet. Кодировка кириллицы и прочих символов **ср-251** используется в Microsoft Windows — см. табл. П2.4. Как видно из этой таблицы, для графических приложений символы псевдографики оказались НЕ нужными.

Таблица П2.3. Кодировка кириллицы и символов псевдографики IBM ср-866 (128-255)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	А	160	A0	а	192	C0	└	224	E0	р
129	81	Б	161	A1	б	193	C1	┴	225	E1	с
130	82	В	162	A2	в	194	C2	┬	226	E2	т
131	83	Г	163	A3	г	195	C3	├	227	E3	у
132	84	Д	164	A4	д	196	C4	─	228	E4	ф
133	85	Е	165	A5	е	197	C5	┼	229	E5	х
134	86	Ж	166	A6	ж	198	C6	╞	230	E6	ц
135	87	З	167	A7	з	199	C7	╟	231	E7	ч
136	88	И	168	A8	и	200	C8	╚	232	E8	ш
137	89	Й	169	A9	й	201	C9	╔	233	E9	щ
138	8A	К	170	AA	к	202	CA	╩	234	EA	ъ
139	8B	Л	171	AB	л	203	CB	╦	235	EB	ы
140	8C	М	172	AC	м	204	CC	╠	236	EC	ь
141	8D	Н	173	AD	н	205	CD	═	237	ED	э
142	8E	О	174	AE	о	206	CE	╬	238	EE	ю
143	8F	П	175	AF	п	207	CF	╧	239	EF	я
144	90	Р	176	B0	░	208	D0	╨	240	F0	Ё
145	91	С	177	B1	▒	209	D1	╤	241	F1	ё
146	92	Т	178	B2	▓	210	D2	╥	242	F2	Є
147	93	У	179	B3	│	211	D3	╙	243	F3	є
148	94	Ф	180	B4	┤	212	D4	╘	244	F4	Ї
149	95	Х	181	B5	╡	213	D5	╒	245	F5	ї
150	96	Ц	182	B6	╢	214	D6	╓	246	F6	Ў
151	97	Ч	183	B7	╖	215	D7	╫	247	F7	ў
152	98	Ш	184	B8	╕	216	D8	╪	248	F8	°
153	99	Щ	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ъ	186	BA	║	218	DA	┌	250	FA	·
155	9B	Ы	187	BB	╗	219	DB	█	251	FB	√
156	9C	Ь	188	BC	╝	220	DC	▄	252	FC	№
157	9D	Э	189	BD	╜	221	DD	▌	253	FD	¤
158	9E	Ю	190	BE	╛	222	DE	▐	254	FE	■
159	9F	Я	191	BF	┐	223	DF	▀	255	FF	

Таблица П2.4. Кодировка кириллицы и прочих символов ср-1251 (128-255).

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ђ	160	A0		192	C0	А	224	E0	а
129	81	Ѓ	161	A1	Ў	193	C1	Б	225	E1	б
130	82	‚	162	A2	ў	194	C2	В	226	E2	в
131	83	ѓ	163	A3	Ј	195	C3	Г	227	E3	г
132	84	„	164	A4	¤	196	C4	Д	228	E4	д
133	85	…	165	A5	Ґ	197	C5	Е	229	E5	е

134	86	†	166	A6	¦	198	C6	Ж	230	E6	ж
135	87	‡	167	A7	§	199	C7	З	231	E7	з
136	88	Ђ	168	A8	Ё	200	C8	И	232	E8	и
137	89	‰	169	A9	©	201	C9	Й	233	E9	й
138	8A	Љ	170	AA	Є	202	CA	К	234	EA	к
139	8B	‹	171	AB	«	203	CB	Л	235	EB	л
140	8C	Њ	172	AC	¬	204	CC	М	236	EC	м
141	8D	Ќ	173	AD	-	205	CD	Н	237	ED	н
142	8E	Ћ	174	AE	®	206	CE	О	238	EE	о
143	8F	Џ	175	AF	Ї	207	CF	П	239	EF	п
144	90	ђ	176	B0	°	208	D0	Р	240	F0	р
145	91	‘	177	B1	±	209	D1	С	241	F1	с
146	92	’	178	B2	І	210	D2	Т	242	F2	т
147	93	“	179	B3	і	211	D3	У	243	F3	у
148	94	”	180	B4	ґ	212	D4	Ф	244	F4	ф
149	95	•	181	B5	µ	213	D5	Х	245	F5	х
150	96	–	182	B6	¶	214	D6	Ц	246	F6	ц
151	97	—	183	B7	·	215	D7	Ч	247	F7	ч
152	98	˜	184	B8	ё	216	D8	Ш	248	F8	ш
153	99	™	185	B9	№	217	D9	Щ	249	F9	щ
154	9A	љ	186	BA	є	218	DA	Ъ	250	FA	ъ
155	9B	›	187	BB	»	219	DB	Ы	251	FB	ы
156	9C	њ	188	BC	ј	220	DC	Ъ	252	FC	ь
157	9D	ќ	189	BD	Ѕ	221	DD	Э	253	FD	э
158	9E	ћ	190	BE	ѕ	222	DE	Ю	254	FE	ю
159	9F	џ	191	BF	ї	223	DF	Я	255	FF	я

Приложение 3. Расширенные ASCII-коды

Расширенные ASCII- коды возвращаются теми клавишами **(функциональные клавиши)** или теми комбинациями клавиш, которые не могут быть представлены стандартными кодами ASCII, перечисленными в таблицах П2.1, П2.2, П2.3 и П2.4.

Расширенный ASCII-код хранится в **двух байтах (первый байт всегда содержит ноль, а второй — scan-код).** В таблице П3.1 приведен второй **(scan)** код в двух системах счисления (**Dec** — десятичная система счисления, **Hex** — шестнадцатеричная система счисления) и нажатию какой клавиши он соответствует (**Key**).

Таблица П3.1. Scan-коды функциональных клавиш.

Key	Hex	Dec	Key	Hex	Dec	Key	Hex	Dec	Key	Hex	Dec
F1	3B	59	Shift-F1	54	84	Ctrl-F1	5E	94	Alt-F1	68	104
F2	3C	60	Shift-F2	55	85	Ctrl-F2	5F	95	Alt-F2	69	105
F3	3D	61	Shift-F3	56	86	Ctrl-F3	60	96	Alt-F3	6A	106
F4	3E	62	Shift-F4	57	87	Ctrl-F4	61	97	Alt-F4	6B	107
F5	3F	63	Shift-F5	58	88	Ctrl-F5	62	98	Alt-F5	6C	108
F6	40	64	Shift-F6	59	89	Ctrl-F6	63	99	Alt-F6	6D	109

Key	Hex	Dec	Key	Hex	Dec	Key	Hex	Dec	Key	Hex	Dec
F7	41	65	Shift-F7	5A	90	Ctrl-F7	64	100	Alt-F7	6E	110
F8	42	66	Shift-F8	5B	91	Ctrl-F8	65	101	Alt-F8	6F	111
F9	43	67	Shift-F9	5C	92	Ctrl-F9	66	102	Alt-F9	70	112
F10	44	68	Shift-F10	5D	93	Ctrl-F10	67	103	Alt-F10	71	113

Продолжение табл. П3.1.

Alt-A	1E	30	Alt-N	31	49	Alt-0	81	129
Alt-B	30	48	Alt-O	18	24	Alt-1	78	120
Alt-C	2E	46	Alt-P	19	25	Alt-2	79	121
Alt-D	20	32	Alt-Q	10	16	Alt-3	7A	122
Alt-E	12	18	Alt-R	13	19	Alt-4	7B	123
Alt-F	21	33	Alt-S	1F	31	Alt-5	7C	124
Alt-G	22	34	Alt-T	14	20	Alt-6	7D	125
Alt-H	23	35	Alt-U	16	22	Alt-7	7E	126
Alt-I	17	23	Alt-V	2F	47	Alt-8	7F	127
Alt-J	24	36	Alt-W	11	17	Alt-9	80	128
Alt-K	25	37	Alt-X	2D	45	Alt--	82	130
Alt-L	26	38	Alt-Y	15	21	Alt-=	83	131
Alt-M	32	50	Alt-Z	2C	44			

Продолжение табл.П3.1.

Key	Hex	Dec	Key	Hex	Dec
↓	50	80			
←	4b	75	Ctrl-←	73	115
→	4d	77	Ctrl-→	74	116
↑	48	72			
NUL	03	3			
Shift-Tab	0f	15			
Delete	53	83			
End	4f	79	Ctrl-End	75	117
Home	47	71	Ctrl-Home	77	119
Insert	52	82			
PageDown	51	81	Ctrl- PageDown	76	118
PageUp	49	73	Ctrl- PageUp	84	132
			Ctrl-PrintScreen	72	114

Таблица П3.2. Scan — коды для расширенной (101-клавишной) клавиатуры.

Key	Hex	Dec	Key	Hex	Dec	Key	Hex	Dec
F11	85	133	Alt-Bksp	0e	14	Alt- **K**/	A4	164
F12	86	134	Alt-Enter	1c	28	Alt- **K***	37	55
Shift-F11	87	135	Alt-Esc	01	1	Alt- **K**-	4a	74
Shift-F12	88	136	Alt-Tab	A5	165	Alt- **K**+	4e	78
Ctrl-F11	89	137	Ctrl-Tab	94	148	Alt- **K** Enter	A6	166
Ctrl-F12	8a	138						
Alt-F11	8b	139	Alt-↑	98	152	Ctrl- **K**/	95	149
Alt-F12	8c	140	Alt-↓	A0	160	Ctrl- **K***	96	150
Alt-[	1a	26	Alt-←	9b	155	Ctrl- **K**-	8e	142
Alt-]	1b	27	Alt-→	9d	157	Ctrl- **K**+	90	144
Alt-;	27	39						
Alt-'	28	40	Alt- Delete	A3	163	Ctrl- **K**↑ [8]	8d	141
Alt-‘	29	41	Alt-End	9f	159	Ctrl- **K**5 [5]	8f	143
Alt-\	2b	43	Alt-Home	97	151	Ctrl- **K**↓ [2]	91	145
Alt-,	33	51	Alt-Insert	A2	162	Ctrl- **K** Ins [0]	92	146
Alt-.	34	52	Alt-PageUp	99	153	Ctrl- **K** Del [.]	93	147
Alt-/	35	53	Alt- PageDown	A1	161			

Префикс **K** означает клавишу на цифровой клавиатуре.

Приложение 4. Базовые арифметические типы данных

Компиляторы с алгоритмических языков оперируют с достаточно широким набором арифметических типов данных (см. соответствующие справочные системы — HELP). В данном приложении даются базовые арифметические типы данных для компиляторов с языков C/C++ и Pascal применительно к 16-ти и 32-разрядному программированию (соответственно платформы Win16 и Win32), с которыми мы будем работать. Как видно из табл.П4.1-П4.4, отличия касаются в основном **целочисленных** данных длиной в 16 и 32 бит.

П4.1. Типы данных для C/C++

Константы, которые задают конкретные максимальные и минимальные значения целочисленных и вещественных данных в зависимости от их типа, содержатся в виде *макросов* в специальных заголовочных файлах **limits.h, float.h, values.h** стандартной библиотеки C++. Эти файлы обеспечивают *переносимость программ* на языке C++, т.е. программа, разработанная для операционной системы Unix должна с минимумом доработок работать в любой другой операционной системе, например, в операционной системе MS DOS или Microsoft Windows (теоретически).

Обычно диапазон представления для *целочисленных данных* **int** зависит от той платформы, для которой разрабатывается приложение (Application) — см. Табл. П4.1 и П4.2. Например, для IBM PC *по умолчанию* компилятор **Borland C++ 4.x** устанавливает платформу **Win16 (Windows 3.x)**, которая называется **EasyWin**, компилятор **Borland C++ 3.x** —

MS DOS Standard. Компиляторы **Borland C++ 5.x, Visual C++4.x** (и выше) — **Win32 Console** — 32-х разрядное консольное приложение (окно MS DOS) в операционных системах линейки **Microsoft Windows 9.x/NT/2000.**

Таблица П4.1. Платформа Win16 (EasyWin, MS DOS).

Тип	Длина (бит)	Диапазон допустимых значений
unsigned char	8	0..255
char	8	-128..127
enum	16	-32768..32767
unsigned int	16	0..65535
short int	16	-32768..32767
int	16	-32768..32767
unsigned long	32	0 .. 4 294 967 295
long	32	-2 147 483 648 .. 2 147 483 647
float	32	3.4 e-38 .. 3.4 e+38
double	64	1.7 e-308 .. 1.7 e+308
long double	80	3.4 e-4932 .. 1.1 e+4932

Таблица П4.2. Платформа Win32.

Тип	Длина (бит)	Диапазон допустимых значений
unsigned char	8	0..255
char	8	-128..127
enum	32	-2 147 483 648 .. 2 147 483 647
unsigned int	32	0 .. 4 294 967 295
short int	16	-32768..32767
int	32	-2 147 483 648 .. 2 147 483 647
unsigned long	32	0 .. 4 294 967 295
long	32	-2 147 483 648 .. 2 147 483 647
float	32	3.4 e-38 .. 3.4 e+38
double	64	1.7 e-308 .. 1.7 e+308
long double	80	3.4 e-4932 .. 1.1 e+4932

П4.2. Типы данных для Borland Pascal и Object Pascal (Delphi-5)

Данные для компиляторов **Borland Pascal 7.0** и **Delphi-5** тоже отличаются целочисленными данными. Из сравнения табл. П4.3 и П.4.4 совершенно четко видно, что **Delphi** — 32-разрядная среда программирования, в которую добавлено несколько целочисленных типов данных.

Таблица П4.3. Borland Pascal (MS DOS, Win16).

Тип	Длина (бит)	Диапазон допустимых значений
Byte	8	0..255
ShortInt	8	-128..127
Word	16	0..65535
Integer	16	-32768..32767
LongInt	32	-2 147 483 648 .. 2 147 483 647
Single	32	3.4 e-38 .. 3.4 e+38
Double	64	1.7 e-308 .. 1.7 e+308
Extended	80	3.4 e-4932 .. 1.1 e+4932

Таблица П4.4. Object Pascal(Delphi5 — Win32).

Тип	Длина (бит)	Диапазон допустимых значений
Byte	8	0..255
ShortInt	8	-128..127
Word	16	0..65535
SmallInt	16	-32768..32767
Integer	32	-2 147 483 648 .. 2 147 483 647
LongInt	32	-2 147 483 648 .. 2 147 483 647
LongWord	32	0 .. 4 294 967 295
Cardinal	32	0 .. 4 294 967 295
Int64	64	-2^63..2^63-1
Single	32	3.4 e-38 .. 3.4 e+38
Double	64	1.7 e-308 .. 1.7 e+308
Extended	80	3.4 e-4932 .. 1.1 e+4932

Приложение 5. Эквивалентные директивы в режимах MASM и Ideal

Директивы и оператор SYMTYPE могут быть набраны в ЛЮБОМ регистре. В таблице для наглядности они приведены ПРОПИСНЫМИ буквами.

MASM	Ideal	MASM	Ideal	MASM	Ideal
.186	P186	.CODE	CODESEG	.FARDATA?	UFARDATA
.286	P286	.COMMENT		.LALL	%MACS
.286C	P286N	.CONST	CONST	.LFCOND	%CONDS
.286P	P286N	.CREF	%CREF	.LIST	%LIST
.287	P287	.DATA	DATASEG	.MODEL	MODEL
.386	P386	.DATA?	UDATASEG	%OUT	DISPLAY
.386C	P386N	.ERR	ERR	PAGE	%PAGESIZE
.386P	P386N	.ERR1	ERRIF1	.RADIX	RADIX
.387	P387	.ERR2	ERRIF2	.SALL	%NOMACS
.486	P486	.ERRB	ERRIFB	.SEQ	
.486C	P486N	.ERRDEF	ERRIFDEF	.SFCOND	%NOCONDS
.486P	P486N	.ERRDIF	ERRIFDIF	.STACK	STACK
.487	P487	.ERRDIF1	ERRIFDIF1	SUBTTL	%SUBTTL
.586	P586	.ERRE	ERRIFE	.TFCOND	
.586C	P586N	.ERRIDN	ERRIFDN	TITLE	%TITLE
.586P	P586N	.ERRIDN1	ERRIFDN1	.TYPE	SYMTYPE
.587	P587	.ERRNB	ERRIFNB	.XALL	
.8086	P8086	.ERRNDEF	ERRIFNDEF	.XCREF	%NOCREF
.8087	P8087	.ERRNZ	ERRIF	.XLIST	%NOLIST
.ALPHA	DOSSEG	.FARDATA	FARDATA		

SYMTYPE является оператором.

Приложение 6. Общая схема распределения памяти в MS DOS

Адреса (Hex)	Наименование и описание
Стандартная память (640К)	
00000-003FF	Таблица векторов прерываний (256 четырех байтных адресов)
00400-004FF	Область данных ROM BIOS (**BIOS DATA Area**)
00500-00xxx	Область DOS (**DOS Area**)
00xxx-9FFFF	Память для пользовательских программ (**User RAM**) <=638K
9FC00-9FFFF	Расширение BIOS DATA Area для PS/2 мыши
Верхняя память - UMA (384К)	
A0000-BFFFF	Видеопамять (**Video RAM**)
C0000-DFFFF	Блоки внешнего кода ROM (по 2К)
E0000-EFFFF	свободная область, иногда **System BIOS**.
F0000-FFFFF	**System BIOS** или **flash-ROM** на системной плате.

Приложение 7. Коды операций команд Ассемблера для процессоров iX86

Таблица П7.1. Основные коды операций.

	X0	X1	X2	X3	X4	X5	X6	X7
0x	ADD r/m, r8	ADD r/m, r16	ADD r8, r/m	ADD r16, r/m	ADD AL, im8	ADD AX,im16	PUSH ES	POP ES
1x	ADC r/m, r8	ADC r/m, r16	ADC r8, r/m	ADC r16, r/m	ADC AL, im8	ADC AX,im16	PUSH SS	POP SS
2x	AND r/m, r8	AND r/m, r16	AND r8, r/m	AND r16, r/m	AND AL, im8	AND AX,im16	SEG ES	DAA
3x	XOR r/m, r8	XOR r/m, r16	XOR r8, r/m	XOR r16, r/m	XOR AL, im8	XOR AX,im16	SEG SS	AAA
4x	INC AX	INC CX	INC DX	INC BX	INC SP	INC BP	INC SI	INC DI
5x	PUSH AX	PUSH CX	PUSH DX	PUSH BX	PUSH SP	PUSH BP	PUSH SI	PUSH DI
6x	*PUSHA	*POPA	*BOUND	ARPL	**SEG FS	**SEG GS	**opSize Prefix	**AddrSize Prefix
7x	JO	JNO	JB/JNAE	JNB/JAE	JE/JZ	JNE/JNZ	JBE/JNA	JNBE/JA
8x	ArOp1 r/m, im8	ArOp1 r/m,im16	ArOp2 r/m, im8	ArOp1 r/m, im16	TEST r/m, r8	TEST r/m, r16	XCHG r8, r/m	XCHG r16, r/m
9x	NOP	XCNG AX,CX	XCNG AX,DX	XCNG AX,BX	XCNG AX,SP	XCNG AX,BP	XCNG AX,SI	XCNG AX,DI
Ax	MOV AL,mem8	MOV AX,mem16	MOV mem8,AL	MOV Mem16,AX	MOVSB	MOVSW	CMPSB	CMPSW
Bx	MOV AL,im8	MOV CL,im8	MOV DL,im8	MOV BL,im8	MOV AH,im8	MOV CH,im8	MOV DH,im8	MOV BH,im8
Cx	*ShfOp r/m,im8	*ShfOp r/m,im16	RET near im16	RET near	LES r16,mem32	LDS r16,mem32	MOV mem8,im8	MOV mem8,im8
Dx	ShftOp r8/m8,1	ShftOp r16/m16,1	ShftOp r8/m8,CL	ShftOp r16/m16,CL	AAM	AAD		XLAT
Ex	LOOPNE LOOPNZ	LOOPE LOOPZ	LOOP	JCXZ **JECXZ	IN AL,p8	IN AX,p8	OUT AL,p8	OUT AX,p8
Fx	LOCK		REP REPNE	REPZ REPE	HLT	CMC	Grp1 r8/m8	Grp1 r16/m16
	X0	X1	X2	X3	X4	X5	X6	X7

Продолжение табл. П7.1.

	X8	X9	Xa	Xb	Xc	Xd	Xe	Xf
0x	OR r/m, r8	OR r/m, r16	OR r8, r/m	OR r16, r/m	OR AL, im8	OR AX,im16	PUSH ES	**Extnsn OpCode
1x	SBB r/m, r8	SBB r/m, r16	SBB r8, r/m	SBB r16, r/m	SBB AL, im8	SBB AX,im16	PUSH DS	POP DS
2x	SUB r/m, r8	SUB r/m, r16	SUB r8, r/m	SUB r16, r/m	SUB AL, im8	SUB AX,im16	SEG CS	DAS
3x	CMP r/m, r8	CMP r/m, r16	CMP r8, r/m	CMP r16, r/m	CMP AL, im8	CMP AX,im16	SEG DS	AAS
4x	DEC AX	DEC CX	DEC DX	DEC BX	DEC SP	DEC BP	DEC SI	DEC DI
5x	POP AX	POP CX	POP DX	POP BX	POP SP	POP BP	POP SI	POP DI
6x	*PUSH imm16	*IMUL r/m,im16	*PUSH imm8	*IMUL r/m,im8	*INSB	*INSW	*OUTSB	*OUTSW
7x	JS	JNS	JP/JPE	JNP/JPO	JL/JNGE	JNL/JGE	JLE/JNG	JNLE/JG
8x	MOV r/m, r8	MOV r/m,r16	MOV r8, r/m	MOV r16, r/m	MOV r/m, seg	LEA r16,mem	MOV seg, r/m	POP r/m
9x	CBW	CWD	CALL far	WAIT	PUSHF	POPF	SAHF	LAHF
Ax	TEST AL,mem8	TEST AX,mem16	STOSB	STOSW	LODSB	LODSW	SCASB	SCASW
Bx	MOV AX,im16	MOV CX,im16	MOV DX,im16	MOV BX,im16	MOV SP,im16	MOV BP,im16	MOV SI,im16	MOV DI,im16
Cx	*ENTER im16,im8	*LEAVE	RET far im16	RET far	INT 3	INT Im8	INTO	IRET
Dx	ESC 0 387/486	ESC 1 387/486	ESC 2 387/486	ESC 3 387/486	ESC 4 387/486	ESC 5 387/486	ESC 6 387/486	ESC 7 387/486
Ex	CALL near	JMP near	JMP far	JMP short	IN AL,DX	IN AX,DX	OUT AL,DX	OUT AX,DX
Fx	CLC	STC	CLI	STI	CLD	STD	Grp2 r8/m8	Grp3 r16/m16
	X8	X9	Xa	Xb	Xc	Xd	Xe	Xf

Таблица П7.2. Вторичные коды операций.

	Mod000r/m	Mod001r/m	Mod010r/m	Mod011r/m
ArOp1	ADD	OR	ADC	SBB
ArOp2	ADD			
ShftOp	ROL	ROR	RCL	RCR
Grp1	TEST		NOT	NEG
Grp2	INC	DEC	CALL near	CALL far
Grp3	INC	DEC	CALL near	CALL far

Продолжение табл. П7.2.

	Mod100r/m	Mod101r/m	Mod110r/m	Mod111r/m
ArOp1	AND	SUB	XOR	CMP
ArOp2		SUB		CMP
ShftOp	SHL/SAL	SHR		SAR
Grp1	MUL	IMUL	DIV	IDIV
Grp2	JMP near	JMP far	PUSH	
Grp3				

Условные обозначения:

* — инструкция может выполняться на процессоре НЕ ниже i286;
** — инструкция может выполняться на процессорах НЕ ниже i386 или i486;
r8 — 8-ми разрядный регистр;
r16 — 16-ти разрядный регистр;
seg — сегментный регистр;
mem8 — адрес области памяти длиной в байт;
mem16 — адрес области памяти длиной в слово;
r/m — регистр или адрес памяти;
r/m8 — регистр или адрес области памяти длиной в байт;
r/m16 — регистр или адрес области памяти длиной в слово;
im8 — константа длиной в байт;
im16 — константа длиной в слово;
imm8 — константа длиной в байт для процессора 80286;
imm16 — константа длиной в слово для процессора 80286;
p8 — 8-ми разрядный порт;
near — внутренний адрес;
far — внешний адрес;
short — короткий внутренний адрес (смещение в пределах от —128 до +127).



ЗАМЕЧАНИЕ

Для процессоров, начиная с **i386**, регистры **r16** и адреса **mem16 (r/m16)** могут быть и 32-разрядными — см. п. 12.5.

Приложение 8. Стандартные преобразования арифметических типов данных

В алгоритмическом языке С++ принято, что при вычислении арифметических выражений с данными одного типа результат получается того же типа (что вполне естественно). Например, если **5/2**, то получится *целая величина* **2**, а если **5.0/2.0**, то получится результат типа **double 2.5**. А что же будет, если данные разных типов? Например, если **5.0/2**, то каков будет результат? Оказывается, выполняются *стандартные преобразования данных низшего типа к высшему типу* **до вычислений** *(но с учетом приоритета операций)*. Таким образом, результат операции **5.0/2 = 2.5**. Ниже приведен мой вольный перевод HELP Borland C++ на эту тему.

Если Вы используете арифметическое выражение типа **a # b**, где **a** и **b** — различные арифметические базовые типы, а # — арифметическая операция, то **Borland C++ преобразовывает операнды прежде, чем вычисляет выражение**. Эти стандартные преобразования включают поддержки «более низких» типов к «более высоким» типам в интересах точности и непротиворечивости.

Borland C++ использует следующие шаги, чтобы преобразовать операнды в арифметическом выражении:

1. Любые меньшие интегральные типы преобразовываются, как показано в табл.П8.1. После этого, любые два значения, связанные с оператором, являются или **int** (включая **long** и без знака **unsigned**), или они имеют тип **double, float** или **long double**.

Таблица П8.1. Преобразование целочисленных данных.

Исходный тип	Результирующий тип	Метод преобразования
char	**int**	Ноль или расширение знака (зависит от значения по умолчанию **char**)
unsigned char	**int**	Заполненный нолем старший байт (всегда)
signed char	**int**	Расширение знака (всегда)
short	**int**	Само значение; расширенный знак
unsigned short	**unsigned int**	Само значение; знаковый разряд – ноль
enum	**int**	Само значение

2. Если один операнд имеет тип **long double**, другой операнд преобразовывается в **long double.**
3. Если один операнд имеет тип **double**, другой операнд преобразовывается в **double**.
4. Если один операнд имеет тип **float**, другой операнд преобразовывается в **float**.
5. Если один операнд имеет тип **unsigned long**, другой операнд преобразовывается в **unsigned long**.
6. Если один операнд имеет тип **long**, то другой операнд преобразовывается в **long**.
7. Если один операнд имеет тип **unsigned**, то другой операнд преобразовывается в **unsigned**.
8. Иначе, оба операнда имеют тип **int**.

Таким образом, результат выражения соответствует самому старшему типу из двух операндов.

Назначение объекта **signed char** интегральному объекту заключается в автоматическом расширении знака. Объекты типа **signed char**, всегда используют расширение знака; объекты типа **unsigned char** всегда устанавливают старший байт в ноль при преобразовании в **int**.

Преобразование более длинного интегрального типа к более короткому типу усекает старшие биты и оставляет неизменными младшие биты. Преобразование короткого ин-

тегрального типа к более длинному типу или расширяет знак или заполняет старшие биты нолями в зависимости от того, знаковый или беззнаковый исходный тип соответственно.



ЗАМЕЧАНИЕ

Все, что здесь написано, справедливо, в основном, и для **Borland Pascal-7.0x**, в тех случаях, когда синтаксис языка допускает вычисление смешанных выражений.

Приложение 9. Коды операций команд сопроцессора

Все коды операций команд сопроцессора начинаются с двоичного набора **11011** (ESC) и имеют длину 2 байта, кроме команды **FWAIT** (ее длина один байт) и команд **FXSTOR** m и **FXSAVE** m (их длина 3 байта). Условимся называть первый байт команды байтом кода операции, а второй байт — либо байтом способа адресации, либо байтом вторичного кода операции в зависимости от типа команды. Назначение и формат команд сопроцессора см. в главе 13.

Для команд сопроцессора, не имеющих операндов, второй байт всегда содержит вторичный код операции.

Для команд сопроцессора, которые имеют операнды (*арифметические команды, команды пересылки данных и управления*), второй байт команд сопроцессора содержит информацию о режиме адресации (по аналогии с байтом способа адресации команд процессора). Как она расшифровывается — см. п. 13.6. Для таких команд, если операнды располагаются в оперативной памяти, в приведенной ниже таблице дан только собственно вторичный код операции — поле $КОП_2$ (биты 3-5), -например, команда **FADD** m32. Если операндами являются регистры сопроцессора **St(i)** (**i=1,...,7**), то во втором байте нужно учитывать номер этого регистра **i** (биты 0-2), — например, команда **FADD** St(0),st(i). Периодически (как пояснение и напоминание) эта информация повторяется и в примечании.

Команда	Байт кода операции	Байт способа адресации / Байт вторичного кода операции	Примечание
F2XM1	D9	F0	
FABS	D9	E1	
FADD m32	D8	0	0 - содержимое поля $КОП_2$
FADD m64	DC	0	
FADD St(0),st(i)	D8	C0i	i - содержимое битов 0-2.
FADD st(i),St(0)	DC	C0i	
FADDP st(i),St(0)	DE	C0i	
FBLD m80	DF	4	4 - содержимое поля $КОП_2$

FBSTP m80	DF	6	
FCHS	D9	E0	
FCLEX	DB	E2	Команда всегда имеет префикс 9B
FCMOVB St(0),st(i)	DA	C0i	Процессор P6
FCMOVE St(0),st(i)	DA	C8i	Процессор P6
FCMOVBE St(0),st(i)	DA	D0i	Процессор P6
FVMOVU St(0),st(i)	DA	D8i	Процессор P6
FCMOVNB St(0),st(i)	DB	C0i	Процессор P6
FCMOVNE St(0),st(i)	DB	C8i	Процессор P6
FCMOVNBE St(0),st(i)	DB	D0i	Процессор P6
FVMOVNU St(0),st(i)	DB	D8i	Процессор P6
FCOM m32	D8	2	2 – содержимое поля $КОП_2$
FCOM m64	DC	2	
FCOM st(i)	D8	D0i	i – содержимое битов 0-2.
FCOMP m32	D8	3	
FCOMP m64	DC	3	
FCOMP st(i)	D8	D8i	
FCOMPP	DE	D9	
FCOMI St(0),st(i)	DB	F0i	Процессор P6
FCOMIP St(0),st(i)	DF	F0i	Процессор P6
FCOS	D9	FF	
FDECSTP	D9	F6	
FDIV m32	D8	6	6 – содержимое поля $КОП_2$
FDIV m64	DC	6	
FDIV St(0),st(i)	D8	F0i	i – содержимое битов 0-2.
FDIV st(i),St(0)	DC	F8i	
FDIVP st(i),St(0)	DE	F8i	
FDIVR m32	D8	7	7 – содержимое поля $КОП_2$
FDIVR m64	DC	7	
FDIVR St(0),st(i)	D8	F8i	i – содержимое битов 0-2.
FDIVR st(i),St(0)	DC	F0i	
FDIVRP st(i),St(0)	DE	F0i	

FDISI	DB	E1	Команда всегда имеет префикс 9B
FENI	DB	E0	Команда всегда имеет префикс 9B
FFREE st(i)	DD	C0i	
FIADD m16	DE	0	0 – содержимое поля $КОП_2$
FIADD m32	DA	0	
FICOM m16	DE	2	2 – содержимое поля $КОП_2$
FICOM m32	DA	2	
FICOMP m16	DE	3	3 – содержимое поля $КОП_2$
FICOMP m32	DA	3	
FIDIV m16	DE	6	6 – содержимое поля $КОП_2$
FIDIV m32	DA	6	
FIDIVR m16	DE	7	7 – содержимое поля $КОП_2$
FIDIVR m32	DA	7	
FILD m16	DF	0	0 – содержимое поля $КОП_2$
FILD m32	DB	0	
FIMUL m16	DE	1	1 – содержимое поля $КОП_2$
FIMUL m32	DA	1	
FINCSTP	D9	F7	
FINIT	DB	E3	Команда всегда имеет префикс 9B
FIST m16	DF	2	2 – содержимое поля $КОП_2$
FIST m32	DB	2	
FISTP m16	DF	3	3 – содержимое поля $КОП_2$
FISTP m32	DB	3	

FISUB m16	DE	4	4 - содержимое поля $КОП_2$
FISUB m32	DA	4	
FISUBR m16	DE	5	5 - содержимое поля $КОП_2$
FISUBR m32	DA	5	
FLD m32	D9	0	0 - содержимое поля $КОП_2$
FLD m64	DD	0	
FLD m80	DB	5	
FLD st(i)	D9	C0i	i - содержимое битов 0-2.
FLD1	D9	E8	
FLDL2T	D9	E9	
FLDL2E	D9	EA	
FLDPI	D9	EB	
FLDLG2	D9	EC	
FLDLN2	D9	ED	
FLDZ	D9	EE	
FLDCW m16	D9	5	5 - содержимое поля $КОП_2$
FLDENV m14	D9	4	
FMUL m32	D8	1	1 - содержимое поля $КОП_2$
FMUL m64	DC	1	
FMUL St(0),st(i)	D8	C8i	i - содержимое битов 0-2.
FMUL st(i),St(0)	DC	C8i	
FMULP st(i),St(0)	DE	C8i	
FNCLEX	DB	E2	
FNDISI	DB	E1	
FNENI	DB	E0	
FNINIT	DB	E3	
FNOP	D9	D0	
FNSAVE m	DD	6	6 - содержимое поля $КОП_2$
FNSETPM	DB	E4	
FNSTCW m16	D9	7	7 - содержимое поля $КОП_2$

FNSTENV m	D9	6	
FNSTSW m16	DD	7	
FNSTSW AX	DF	E0	
FPATAN	D9	F3	
FPREM	D9	F8	
FPREM1	D9	F5	
FPTAN	D9	F2	
FRNDINT	D9	FC	
FRSTOR m	DD	4	4 – содержимое поля $КОП_2$
FSAVE m	DD	6	Команда всегда имеет префикс 9B
FSCALE	D9	FD	
FSETPM	DB	E4	Команда всегда имеет префикс 9B
FSIN	D9	FE	
FSINCOS	D9	FB	
FSQRT	D9	FA	
FST m32	D9	2	2 – содержимое поля $КОП_2$
FST m64	DD	2	
FST st(i)	DD	D0i	i – содержимое битов 0-2.
FSTP m32	D9	3	3 – содержимое поля $КОП_2$
FSTP m64	DD	3	
FSTP m80	DB	7	
FSTP st(i)	DD	D8i	i – содержимое битов 0-2.
FSTCW m16	D9	7	Команда всегда имеет префикс 9B
FSTENV m	D9	6	Команда всегда имеет префикс 9B
FSTSW m16	DD	7	Команда всегда имеет префикс 9B
FSTSW AX	DF	E0	Команда всегда имеет префикс 9B

FSUB m32	D8	4	4 - содержимое поля $КОП_2$
FSUB m64	DC	4	
FSUB St(0),st(i)	D8	E0i	i - содержимое битов 0-2.
FSUB st(i),St(0)	DC	E8i	
FSUBP st(i),St(0)	DE	E8i	
FSUBR m32	D8	5	5 - содержимое поля $КОП_2$
FSUBR m64	DC	5	
FSUBR St(0),st(i)	D8	E8i	i - содержимое битов 0-2.
FSUBR st(i),St(0)	DC	E0i	
FSUBRP st(i),St(0)	DE	E0i	
FTST	D9	E4	
FUCOM st(i)	DD	E0i	i - содержимое битов 0-2.
FUCOMP st(i)	DD	E8i	
FUCOMPP	DA	E9	
FUCOMI St(0),st(i)	DB	E8i	Процессор P6
FUCOMIP St(0),st(i)	DF	E8i	Процессор P6
FWAIT	9B		
FXAM	D9	E5	
FXCH st(i)	D9	C8i	i - содержимое битов 0-2.
FXSTOR m	OF	AE/1	Процессор P6
FXTRACT	D9	F4	
FXSAVE m	OF	AE/0	Процессор P6
FYL2X	D9	F1	
FYL2XP1	D9	F9	

Список использованной и рекомендуемой литературы

Надо пользоваться не красотой книг и не их количеством, но их речью и всем, что в них написано.

Лукиан (ок. 120 г. — ок. 190 г.)

Л1. Немного истории

1. Гутер Р.С., Полунов Ю.Л. От абака до компьютера. — 2-е изд., испр. и доп. — М.: Знание, 1981.
2. Данилов Ю.А. Джон фон Нейман. — М.: Знание, 1981.
3. Дашевский Л.Н., Шкабара Е.А. Как это начиналось (Воспоминания о создании первой отечественной электронной вычислительной машины — МЭСМ). — М.: Знание, 1981.
4. Касперски К. Техника сетевых атак. — М.: Солон-Р, 2001.
5. Кэпс Ч., Стаффорд Р. VAX: Программирование на языке ассемблера и архитектура. — М.: Радио и связь, 1991 .
6. Левитин К.Е. Прощание с АЛГОЛом. — М.: Знание, 1989.
7. Лин В. PDP-11 и VAX-11. Архитектура ЭВМ и программирование на языке ассемблера. — М.: Радио и связь, 1989.
8. Страуструп Б. Дизайн и эволюция языка С++. — М.: ДМК Пресс, 2000.
9. Стэбли Д. Логическое программирование в системе /360. — М.: Мир, 1974.
10. Фокс Дж. Программное обеспечение и его разработка. — М.: Мир, 1985.

Л2. Как работает компьютер

1. Гук М., Юров В. Процессоры Pentium III, Athlon и другие. — СПб.: Питер, 2000.
2. Знакомьтесь: компьютер. — М.: Мир, 1989.
3. Ледли Р. Программирование и использование вычислительных машин. — М.: Мир, 1966.
4. Нортон П., Гудмен Дж. Работа на персональном компьютере. Самоучитель. — К.: ДиаСофт, 1999.
5. Петцольд Ч. Код. Тайный язык информатики. — М.: Издательско-торговый дом "Русская редакция", 2001.
6. Хасэгава Х. Мир компьютеров в вопросах и ответах: В 2-х кн. — М.: Мир, 1988.
7. Язык компьютера. — М.: Мир, 1989.

Л3. Новые технологии программирования

1. Кастер Х. Основы Windows NT и NTFS. — М.: Издательский отдел Русская редакция ТОО "Channel Trading Ltd.", 1996.
2. Соломон Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000. Мастер-класс. — СПб.: Питер; М.: Издательско-торговый дом "Русская редакция", 2001.
3. Трельсен Э. Модель COM и применение ATL 3.0. — СПб.: BHV-Санкт-Петербург, 2000.

Л4. Assembler IBM PC

1. Абель П. Язык ассемблера для IBM PC и программирования. — М.: Высшая школа, 1992.
2. Бердышев Е. Технология MMX. Новые возможности процессоров P5 и P6. — М.: ДИАЛОГ-МИФИ, 1998.
3. Брэдли Д. Программирование на языке ассемблера для персональной ЭВМ фирмы IBM. — М.: Радио и связь, 1988.
4. Голубь Н.Г., Пудовкина Л.Ф. Методические рекомендации по выполнению лабораторных работ по курсу "Организация и функционирование ЭВМ". — Харьков: ХАИ, 1993.
5. Григорьев В.Л. Архитектура и программирование арифметического сопроцессора.- М.: Энергоатомиздат, 1991.
6. Джордейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT и AT. — М.: Финансы и статистика, 1992.
7. Зубков С.В. Assembler для DOS, Windows и Unix. — 2-е изд., испр. и доп. — М.: ДМК, 2000.
8. Использование Turbo Assembler при разработке программ. — К.: Диалектика, 1994.

9. Нортон П. Персональный компьютер фирмы IBM и операционная система MS-DOS.- М.: Радио и связь, 1991.
10. Нортон П. Программно-аппаратная организация IBM PC.- М.: Радио и связь, 1991.
11. Нортон П., Соухэ Д. Язык ассемблера для IBM PC. — М.: Компьютер, 1993.
12. Оператору ЭВМ. DOS & BIOS.- М.: Наука, 1990.
13. Пильщиков В.Н. Программирование на языке ассемблера IBM PC.- М.: ДИАЛОГ-МИФИ, 1999.
14. Пирогов В.Ю. ASSEMBLER. Учебный курс. — М.: Нолидж, 2001.
15. Ровдо А.А. Микропроцессоры от 8086 до Pentium III Xeon и AMD-K6-3. — М.: ДМК, 2000.
16. Рудаков П.И., Финогенов К.Г. Язык ассемблера: уроки программирования. — М.: ДИАЛОГ-МИФИ, 2001.
17. Сван Т. Освоение Turbo Assembler. — К.: Диалектика, 1996.
18. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера. — М.: Радио и связь, 1989.
19. Юров В. Assembler. Учебник. — СПб.: Питер, 2000.
20. Юров В. Assembler: практикум. — СПб.: Питер, 2001.
21. Юров В. Assembler: специальный справочник. — СПб.: Питер, 2000.

Л5. Pascal

1. Абрамов С.А., Зима Е.В. Начала информатики. — М.: Наука. Гл.ред.физ.-мат.лит., 1989.
2. Вирт Н. Алгоритмы + структуры данных = программы. — М.: Мир, 1985.
3. Голубь Н.Г., Кириленко Е.Г. Алгоритмические языки и программирование: Учебное пособие по выполнению контрольных и лабораторных работ, часть 1. — Харьков: ХАИ, 1997.
4. Голубь Н.Г., Кириленко Е.Г. Алгоритмические языки и программирование: Методические рекомендации по выполнению контрольных и лабораторных работ, часть 2. — Харьков: ХАИ, 1998.
5. Голубь Н.Г., Кириленко Е.Г. Алгоритмические языки и программирование: Методические рекомендации по выполнению контрольных и лабораторных работ, часть 3. — Харьков: ХАИ, 1998.
6. Грогоно П. Программирование на языке Паскаль. — М.: Мир, 1982.
7. Джонс Ж., Харроу К. Решение задач в системе Турбо Паскаль. — М.: Финансы и статистика, 1991.
8. Емелина Е.И. Основы программирования на языке ПАСКАЛЬ. — М.: Финансы и статистика, 1997.

9. Йенсен К., Вирт Н. Паскаль: руководство для пользователя. — М.: Финансы и статистика, 1989.

10. Касьянов В.Н., Сабельфельд В.К. Сборник заданий по практикуму на ЭВМ. — М.: Наука. Гл.ред.физ.-мат.лит., 1986.

11. Мизрохи С.В. TURBO PASCAL и объектно-ориентированное программирование. — М.: Финансы и статистика, 1992.

12. Пильщиков В.Н. Сборник упражнений по языку Паскаль. — М.: Наука. Гл.ред.физ.-мат.лит., 1989.

13. Прайс Д. Программирование на языке Паскаль: Практическое руководство. — М.: Мир, 1987.

14. Фаронов В.В. Турбо Паскаль 7.0. Начальный курс. Учебное пособие. — М.: Нолидж, 1997.

15. Фаронов В.В. Турбо Паскаль 7.0. Практика программирования. Учебное пособие. — М.: Нолидж, 1997.

Л6. C/C++

1. Аммерааль Л. STL для программистов на С++. — М.: ДМК, 1999.

2. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++, 2-е изд. — М.: Издательство БИНОМ, СПб.: Невский диалект, 1998.

3. Голубь Н.Г. Объектно-ориентированное программирование: Методические рекомендации по выполнению контрольных и лабораторных работ. — Харьков: ХАИ, 2000.

4. Дейтел Х.М., Дейтел П.Дж. Как программировать на С++: второе издание. — М.: ЗАО Издательство БИНОМ, 1999.

5. Дейтел Х., Дейтел П. Как программировать на С++: третье издание. — М.: ЗАО Издательство БИНОМ, 2001.

6. Керниган Б., Ритчи Д. Язык программирования Си. Задачи по языку Си. — М. : Финансы и статистика, 1985.

7. Киммел П. и др. Borland С++ 5. — СПб.: BHV-Санкт-Петербург, 1997.

8. Крэйг А. Borland С++ 5: Освой самостоятельно. — М.: Восточная Книжная Компания, 1997.

9. Либерти Дж.. С++. Энциклопедия пользователя. — К.: ДиаСофт, 2000.

10. Липпман С.Б., Лажойе Ж. Язык программирования С++. Вводный курс, 3-е изд. — СПб., М.: Невский Диалект — ДМК Пресс, 2001.

11. Прата С. Язык программирования С++. Лекции и упражнения. Учебник. — К.: ДиаСофт, 2001.

12. Справочник по классам Borland С++ 4.0. — К.: Диалектика, 1994.

13. Справочник по функциям Borland С++ 3.1/4.0. — К.: Диалектика, 1994.

14. Страуструп Б. Язык программирования C++, спец. изд. — М.; СПб.: Издательство БИНОМ — Невский Диалект, 2001.

15. Страуструп Б. Язык программирования C++. 3-е изд. — СПб.; М.: Невский Диалект — Издательство БИНОМ, 1999.

16. Страуструп Б. Язык программирования C++. 2-я ред. Части 1, 2. — К.: ДиаСофт, 1993.

17. Сэвитч У. C++ в примерах. — М.: БИНОМ, 1997.

18. Уэйт М., Прата С., Мартин Д. Язык Си. Руководство для начинающих. — М.: Мир, 1988.

19. Хэзфилд Р., Кирби Л. и др. Искусство программирования на С. Фундаментальные алгоритмы, структуры данных и примеры приложений. Энциклопедия программиста. — К.: ДиаСофт, 2001.

20. Цимбал А.А., Майоров А.Г., Козодаев М.А. Turbo C++: язык и его применение. — М.: Джен Ай Лтд, 1993.

21. Шилдт Г. Самоучитель C++. 3-е изд. — СПб.: BHV-Санкт-Петербург, 1998.

22. Шилдт Г. Справочник программиста C/C++. — М.: Издательский дом Вильямс, 2000.

23. Шилдт Г. Теория и практика C++. — СПб.: BHV-Санкт-Петербург, 1999.

24. Элджер Дж. C++: библиотека программиста. — СПб.: ЗАО "Издательство Питер", 1999.

Л7. Windows-программирование

1. Гилберт С., Маккарти Б. Программирование на Visual C++ 6. Этюды профессионалов. — К.: Издательство ДиаСофт, 1999.

2. Грегори К. Использование Visual C++ 6. К.: Диалектика, 1999.

3. Культин Н. Delphi 4. Программирование на Object Pascal. — СПб.: БХВ — Санкт-Петербург, 1999.

4. Кэнту М. Delphi 4 для профессионалов. — СПб.: Издательство Питер, 1999.

5. Мешков А., Тихомиров Ю. Visual C++ и MFC. Программирование для Windows NT и Windows 95: В трех томах. СПб.: БХВ — Санкт-Петербург, 1997.

6. Нортон П., Макгрегор Р. Windows 95/NT4. Программирование с помощью MFC. Кн.1,2. — М.: Издательство СК Пресс, 1998.

7. Паппас К.Х., Мюррей У.Х. III. Отладка в C++. — М.: ЗАО "Издательство БИНОМ", 2001.

8. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows. 4-е изд. — СПб.: Питер; М.: Издательско-торговый дом Русская редакция, 2001.

9. Роббинс Д. Отладка Windows-приложений. — М.: ДМК Пресс, 2001.

10. Румянцев П.В. Азбука программирования в Win32 API. 2-е изд. — М.: Радио и связь, Горячая линия — Телеком, 1999.
11. Румянцев П.В. Работа с файлами в Win32 API. 2-е изд. — М.: Горячая линия — Телеком, 2000.
12. Саймон Р. Windows 2000 API. Энциклопедия программиста. — К.: ДиаСофт, 2001.
13. Секунов Н. Самоучитель Visual C++ 6. — СПб.: БХВ — Санкт-Петербург, 1999.
14. Тихомиров Ю. Visual C++ 6. Новые возможности для программистов. — СПб.: БХВ — Санкт-Петербург, 1998.
15. Тосс В. Visual C++ 5. Энциклопедия пользователя. К.: Издательство ДиаСофт, 1998.
16. Фаронов В.В. Delphi 4. Учебный курс. — М.: Нолидж, 1999.
17. Холзнер С. Microsoft Visual C++ 5 с самого начала. СПб., 1998.
18. Холингвэрт Дж., Баттерфилд Д., Сворт Б. и др. C++ Builder 5. Руководство разработчика: В 2-х томах. — М.: Издательский дом Вильямс, 2001.
19. Шамис В.А. C++ Builder 5. Техника визуального программирования. 3-е изд. — М.: Нолидж, 2001.

Л8. Электронные источники информации

1. Журнал PC Magazine/Russian Edition Online — http://www.pcmag.ru/.
2. Журнал PCWeek/Russian Edition Online — http://www.pcweek.ru.
3. Журнал ZDNet Россия — http://zdnet.ru/.
4. Информация для разработчиков на русском языке — http://developer.intel.ru/design/.
5. Официальный сайт компании Microsoft http://www.microsoft.com.
6. Официальный сайт фирмы AMD, содержащий информацию для разработчиков — http://www.amd.com:7246/us-en/Processors/DevelopWithAMD/
7. Официальный сайт фирмы Intel, содержащий документацию для разработчиков IntelR PentiumR 4 Processors — Manuals — http://developer.intel.com/design/pentium4/manuals/.
8. Сайт "Russian Software Developer Network — RSDN" содержит разнообразную информацию для разработчиков программного обеспечения на русском языке http://www.rsdn.ru.
9. Сайт "Программирование на Ассемблере под Win32" — http://hi-tech.nsys.by/.
10. Сайт автора данной книги Язык C++ для начинающих — http://www.anriintern.com/computer/c++/.
11. Сайт на английском языке "Iczelion's Win32 Assembly Homepage" содержит очень много полезной информации, в том числе и очень подробный учебник по программированию на Ассемблере в Win32 http://win32asm.cjb.net/
12. Сайт на русском языке "Windows Assembly Site" http://www.wasm.boom.ru/.

Глоссарий

Пожалуйста, продолжайте изучать персональный компьютер и все другое, чем вы еще интересуетесь. Это лучший способ поддерживать себя умственно живым и заинтересованным в себе и других.

Питер Нортон

К этой части книги следует обращаться, если вы встретили незнакомое слово или фразу (особенно в документации на английском языке). Дополнительные разъяснения можно получить в тексте книги через предметный указатель.

3DNow! — дополнение к обычному набору инструкций для процессоров iX86, введенное компанией AMD и теперь поддерживаемое несколькими производителями процессоров. Команды 3DNow идейно подобны командам MMX, но наряду с данными и командами плавающей арифметики поддерживают и целочисленные данные и команды. Эти команды предназначены для ускорения обработки игровых программ, интенсивно работающих с трехмерной графикой. Они поддерживаются набором драйверов Microsoft DirectX версий 6.0 и выше.

Abort (аварийное завершение программы, сброс) — исключение (особая ситуация), при котором дальнейшее нормальное выполнение программы невозможно. См. **Exception.**

Address (адрес) — см. **Effective Address, Logical Address**, **Linear Address** и **Physical Address**.

Address Space (адресное пространство) — множество ячеек памяти, к которым можно получить доступ по адресу.

Address-Size Prefix (префикс размера адреса) — префикс команды, который определяет размер используемых смещений адресов. Смещение может иметь размер 16 или 32 бита. Размер адреса по умолчанию (т.е. при отсутствии префикса) определяется битом **D** для каждого кодового сегмента. Использование префикса позволяет задать размер операндов, отличный от принятого по умолчанию.

AGP (Advanced Graphics Port — расширенный графический порт, графический ускоритель) — шина, соединяющая видеопроцессор с основной памятью и работающая на полной скорости системной шины.

ALU (Arithmetic Logic Unit — арифметико-логическое устройство, **АЛУ**) — компонент процессора, отвечающий за выполнением целочисленных и логических операций.

Base

1. Основание — термин, используемый в логарифмических и степенных функциях. В обоих контекстах это число, возводимое в степень
2. Основание системы счисления — число, определяющее представление чисел в виде строк цифр.
   - Основание 2 — двоичная система счисления;
   - Основание 10 — десятичная система счисления;
   - Основание 16 — шестнадцатеричная система счисления.
3. Базовый адрес — см. **Base Address**.
4. Базовый регистр — см. **Base Register**.

Base Address (базовый адрес, база) — начальный адрес некоторой структуры данных, например, такой как сегмент, страница или таблица страниц.

Base Register (базовый регистр, регистр базы) — регистр, используемый для вычисления адреса операнда, смещенного относительно адреса, хранимого в этом регистре.

Binary Point (двоичная точка) — точка, подобная десятичной, но используемая в двоичном представлении вещественного числа с плавающей точкой. Каждая двоичная цифра справа от точки умножается на отрицательную степень двойки, а слева — на положительную или ноль степень двойки в зависимости от занимаемой цифрой позиции.

Bit Field (битовое поле) — последовательность до 32 смежных битов, которая может начинаться с любого бита в байте. Для эффективной работы с битовыми полями в процессоре, начиная с i386, предусмотрены специальные операции.

Bit String (битовая строка) — последовательность смежных битов длиною до 2^{32}-1 (4 Г) байт, которая может начинаться с любого бита в байте. Для эффективной работы с битовыми строками в процессоре, начиная с i386, предусмотрены специальные операции.

Break Point (контрольная точка, точка останова или прерывания) — используется для отладки программ. С помощью этих точек программист может определить формы доступа к памяти, которые будут генерировать специальное исключение. Это исключение вызывает программный отладчик. Процессор поддерживает аппаратные и программные контрольные точки. При выполнении команды INT n происходит программное прерывание с номером n. Аппаратные контрольные точки устанавливаются программистом с помощью регистров отладки. Содержимое этих регистров определяет адрес, размер и тип ссылок не более, чем для четырех контрольных точек. В отличие от программных контрольных точек, аппаратные точки прерывания могут быть определены и для данных.

Byte (байт) — 8-ми битовая ячейка памяти. Наименьшая адресуемая единица памяти.

Cache (сверхоперативная память, кэш) — быстрая память малого объема, в которой хранятся значения активных элементов основной «медленной» памяти.

Cache Flush (очистка кэша) — операция, при которой все строки кэша помечаются как недостоверные.

Cache Line (строка кэша) — наименьшая единица информации, которая может быть размещена в кэше.

Cache Line Fill (заполнение строки кэша) — операция загрузки полной строки кэша с использованием многочисленных обращений к основной памяти для считывания.

Call Gate (шлюз вызова) — дескриптор шлюза для вызова процедур при помощи команды CALL или JMP.

Characteristic (характеристика) — термин, означающий смещенный на определенное число порядок вещественного числа с плавающей точкой.

Code Segment (сегмент кода, кодовый сегмент) — адресное пространство, содержащее команды; исполняемый сегмент. Тип информации, хранимой в сегменте, определяется в дескрипторе сегмента (начиная с i386).

Condition Code (код условия) — 4 бита кода условия в слове состояния сопроцессора, в которые заносятся значения при сравнении, тестировании, анализе и других функциях модуля обработки операций с плавающей точкой.

Control Word (управляющее слово, слово управления) — 16-битовый регистр модуля обработки операций с плавающей точкой, с помощью которого пользователь может определить режим вычислений и замаскировать прерывания при исключениях.

Coprocessor (сопроцессор) — расширение основной архитектуры и множества команд процессора. Арифметический (математический) сопроцессор используется для дополнения возможностей процессора новыми арифметическими командами для работы с вещественными числами и новыми регистрами. Начиная с процессора i486DX, сопроцессоры стали встраиваться в микросхему центрального процессора.

CPL (Current Privilege Level — текущий уровень привилегированности) — уровень привилегированности выполняющейся программы. Обычно уровень привилегированности загружается из дескриптора сегмента кода.

CPU (Central Processing Unit — центральное процессорное устройство) — см. **Processor**.

Data Segment (сегмент данных) — адресное пространство, в котором находятся данные. Четыре сегмента данных (начиная с процессора i386) могут использоваться одновременно без изменения содержимого сегментных регистров. Тип информации, хранимой в сегменте, определяется в дескрипторе сегмента.

Data Structure (структура данных) — область памяти, предназначенная для определенного использования аппаратными или программными средствами. Например, таблица страниц или сегмент состояния задачи (см. **TSS**).

Debug Registers (регистры отладки) — множество регистров, используемых для четырех контрольных точек прерывания. В отличие от команд задания контрольных точек, которые могут использоваться только для установки контрольных точек в тексте программы, в регистрах отладки можно определить контрольные точки как для текста программы, так и для данных.

Denormal (денормализованное) — специальная форма числа с плавающей точкой. Для модуля обработки операций с плавающей точкой это число, у которого биты экспоненты (характеристики) равны 0. При использовании мантиссы с начальными нулями, диапазон возможных отрицательных порядков может быть расширен за счет битов мантиссы. Каждый начальный нуль — бит потери точности, поэтому расширенный диапазон порядка получается уменьшением значимости.

Descriptor Table (таблица дескрипторов) — массив дескрипторов сегментов. Существует две разновидности таблиц: Global Descriptor Table (глобальная таблица дескрипторов) и Local Descriptor Tables (локальные таблицы дескрипторов).

Device Driver (драйвер устройства) — процедура или задача, используемая для управления периферийным устройством. Например, драйвер диска.

Diplacement (смещение) — константа для вычисления исполнительных адресов. Смещение изменяет значение адреса независимо от масштабированного индексирования. Оно часто используется для доступа к операндам, имеющим фиксированное смещение относительно какого-либо адреса, например, к полю записи в массиве.

DLL (Dinamic Link Library — библиотека динамического связывания) — исполняемый файл, содержащий одну или несколько программ, к которым могут обращаться другие программы. DLL могут загружаться операционной системой только при необходимости или быть загруженными постоянно.

DOS (Disk Operation System — дисковая операционная система) — первоначально названная так операционная система, чтобы отличаться от предшествовавших ей операционных систем, загружавшихся с магнитной ленты или других внешних устройств. Сейчас это операционная система, которая загружается с диска при включении компьютера.

Double Extended (двойной расширенный) — термин стандарта IEEE-754 для расширенного формата данных модуля обработки операций с плавающей точкой. Характеризуется большей разрядностью под порядок и мантиссу, чем обычный формат двойной точности, и наличием в поле мантиссы явного бита ее целой части.

Double Format (двойной формат) — формат для чисел с плавающей точкой, поддерживаемый модулем обработки операций с плавающей точкой. Состоит из 11-разрядной характеристики, неявного (скрытого) бита целой части мантиссы и 52-разрядной мантиссы, всего 64 явных бита.

Double Word (двойное слово) — область памяти, размером 32 бита. Процессор iX86 поддерживает размещение двойного слова с любого байта, однако при пересечении двойным словом границы между двумя двойными словами в физической памяти, производительность процессора при обращении к такому двойному слову может понизиться.

DPL (Descriptor Privilege Level — дескриптор уровня привилегированности) — уровень привилегированности описываемого дескриптором сегмента. DPL — это поле в дескрипторе сегмента.

Effective Address (**EA** — исполнительный адрес) — адрес, получаемый в общем случае путем сложения базового регистра с масштабируемым индексом и смещением.

Environment (среда вычислений) — 14 или 28 (в зависимости от режима адресации) байтов регистров сопроцессора, содержимое которых изменяется при использовании команд FSTENV и FLDENV. Включает управляющее слово, слово состояния, слово признаков (тегов), а также команды, код и информацию об операндах, используемую при обработке прерываний.

ESC Instruction (команда **ESC**) — префикс кода команд сопроцессора.

Exception (исключение, особая ситуация) — вызов процедуры или задачи, который происходит при неудачной попытке процессора выполнить команду или в случае выполнения команды INT n. Исключения возможны при делении на 0, переполнении стека, неопределенных командах и при прерываниях, вызов которых определен программно. Исключения подразделяются на сбои (отказы), ловушки, остановы и программно инициализируемые прерывания.

Exceptions Pointers (ссылки (указатели) исключений) — в модуле управления данными с плавающей точкой указатель, используемый обработчиком исключений для установления причины нарушения. Состоит из ссылки на последнюю выполненную команду сопроцессора и ссылки на операнд памяти, если такой операнд имелся в команде. Для получения значения этих ссылок обработчик исключений может использовать команды FSTENV и FSAVE.

Exponent (экспонента — порядок).

1. Число, указывающее степень, в которую надо возвести другое число.
2. Поле числа с плавающей точкой, которое указывает степень числа (смещенный порядок). Это соответствует более общему определению порядка, данному в 1, за исключением того, что для получения истинного значения степени необходимо из порядка вычесть смещение. См. **Unbiased Exponent.**

Extended Format (расширенный формат) — реализация для сопроцессора двойного расширенного формата стандарта IEEE 754. Расширенный формат — основной рабочий формат представления чисел с плавающей точкой в модуле обработки операций с плавающей точкой. Состоит из знака, 15-разрядной экспоненты и мантиссы (явный бит целой части и 63 бита дробной части числа).

External Cache (внешний кэш) — память вне процессора. Внешний кэш может быть добавлен к любому процессору, имеющему внешнюю память. Начиная с процессора i486, есть команды и биты входа в таблицу страниц для управления внешним кэшем из программы.

Far Pointer (дальняя ссылка) — ссылка на место в памяти, состоящая из селектора сегмента и смещения. Используется для доступа к памяти, если селектор не был загружен в процессор, например, при вызове процедуры из другого кодового сегмента.

Fault (ошибка, сбой, отказ). После того, как произошла ошибка (особая ситуация, исключение), достаточно восстановить состояние процессора, чтобы попытаться еще раз выполнить команду, вызвавшую исключение. Обработчик исключения вызывается с адресом возврата, который указывает на неудачную команду, а не на следующую. После того как обработчик установил причину отказа, программа иногда может продолжить выполнение.

Flash Memory (флэш-память) — тип памяти с произвольным доступом. Эти модули памяти способны хранить данные даже, если на них длительное время не подавать напряжения. Но для чтения данных нужна энергия.

Flat Model (плоская, сплошная или линейная модель) — организация оперативной памяти, при которой все сегменты отображаются на единственную область линейных адресов. Такая организация памяти исключает, насколько это возможно, сегментацию памяти для прикладных программ.

Floating Point Operand (операнд с плавающей точкой) — представление числа следующим образом: основание системы счисления, знак, мантисса и знаковый порядок. Значение числа есть результат произведения мантиссы (со знаком) на основание, возведенное в степень порядка. Представление чисел с плавающей точкой является более гибким, чем представление целых по двум причинам. Во-первых, оно позволяет представлять дроби. Во-вторых, при таком представлении можно получить больший диапазон порядка, чем возможно при использовании целых фиксированной длины.

FPU (Floating Point Unit) — модуль обработки операций с плавающей точкой — см. **Coprocessor**.

Gate Descriptor (дескриптор шлюза) — дескриптор сегмента для описания назначений вызовов и переходов. Дескриптор шлюза может использоваться для вызова процедур или задач с отличным от вызывающей программы уровнем привилегированности (**DPL**). Существует четыре типа дескрипторов шлюза: шлюзы вызова, шлюзы ловушек, шлюзы прерываний и шлюзы задач.

Global Descriptor Table (**GDT** — таблица глобальных дескрипторов) — таблица, которая создается в оперативной памяти прежде, чем любой процессор x86 сможет перейти в защищенный режим. В системе существует только одна таблица глобальных дескрипторов.

Gradual Underflow (последовательное отрицательное переполнение) — метод обработки ошибок, вызванных отрицательным переполнением при выполнении операций с плавающей точкой, который минимизирует потерю точности результата. Так, если есть денормализованное число, являющееся приемлемым в качестве результата, то оно и возвращается в качестве результата. Таким образом, потеря значащих цифр происходит только в процессе денормализации. Большинство ЭВМ при отрицательном переполнении возвращает нуль, то есть теряет все значащие цифры.

Handler (обработчик) — процедура или задача, вызываемые в случае генерации исключения или прерывания.

IEEE Standard 754, 854 (IEEE — Institute for Electrical and Electronic Engineering — Институт инженеров по электротехнике и радиоэлектронике) — стандарты, формально описывающие множество форматов и операций для чисел с плавающей точкой. Форматы имеют размеры операндов 32, 64 и 80 разрядов. Современный сопроцессор обеспечивает полную поддержку этих форматов.

Immediate Operand (непосредственный операнд, константа) — данные, закодированные непосредственно в команде.

Implicit Integer Bit (неявный бит целой части) — часть мантиссы в нормализованном представлении вещественного или двойного вещественного числа с плавающей точкой, которая в формате операнда задана неявно. В этих форматах полной мантиссой считается значение справа от двоичной точки. Неявный бит целой части слева от двоичной точки всегда равен 1, за исключением единственного случая: если порядок минимальный (смещенный порядок есть 0), то неявный бит целой части равен 0. См. также **Integer Bit.**

Indefinite (INF — бесконечность) — специальное значение, возвращаемое функциями с плавающей точкой, если входные данные таковы, что других разумных ответов нет. Для каждого формата чисел с плавающей точкой существует свое NaN (не-число), определенное как бесконечная величина. Для целочисленных двоичных форматов бесконечной величиной часто считают отрицательное число, наиболее отдаленное от нуля.

Index (индекс) — число, используемое для доступа к таблице. Чтобы вычислить размер операнда, индекс может масштабироваться (умножаться при помощи сдвига влево — для процессоров, начиная с i386). Для получения точки входа в таблицу масштабированный индекс прибавляется к базовому адресу таблицы.

Inexact (неточность) — стандартный термин IEEE-754 для обозначения ошибочной точности при обработке операции с плавающей точкой.

Initialization (инициализация) — процесс присваивания начальных значений параметрам программной среды после перезагрузки. Процессор начинает работу после перезагрузки в режиме реальной адресации. Несколько регистров процессора определяют состояние готовности к работе. Начальные состояния сегментных регистров (начиная с i386) позволяют получать доступ к памяти, даже если селекторы сегментов еще не были загружены. Очистка регистра DR7 (регистр управления отладкой) блокирует все прерывания во время инициализации. Программа в реальном режиме адресации может инициализировать такие структуры данных как таблицы дескрипторов и таблицы страниц, а затем перевести процессор в защищенный режим.

Instruction Prefetch (предвыборка команд, выборка команд с упреждением) — загрузка в процессор команд, следующих в памяти непосредственно за текущей, одновременно с ее выполнением. Так называется технология для одновременного (параллельного) выполнения нескольких команд (начиная с i486).

Interface (интерфейс, стык) — совокупность средств и правил, обеспечивающих логическое или физическое взаимодействие устройств и/или программ вычислительной системы. Физический интерфейс определяет тип стыка и его свойства. Программный интерфейс определяет совокупность допустимых процедур или операций и их параметров, список общих переменных, областей памяти или других объектов.

Instruction Restart (перезапуск команды) — возможность сделать повторную попытку выполнения команды, вызвавшей исключение (начиная с i486). Если есть необходимость обращения к сегменту или странице, которые не представлены в памяти, то повторная попытка должна быть выполнена только после того, как операционная система произвела подкачку страницы или сегмента в оперативную память. Перезапуск команды позволяет восстановить состояние процессора, допускающее вызов обработчика исключения с адресом возврата, указывающим на команду, сгенерировавшую исключение, а не на следующую.

Integer (целое) — конечное число (положительное, отрицательное, нуль), не имеющее дробной части. Также может означать представление числа в ЭВМ: последовательность байтов данных, интерпретируемых стандартно. Разумно представлять целые в формате с плавающей точкой, если они превышают допустимый для целочисленных вычислений диапазон. В этом случае их обработкой займется сопроцессор.

Integer Bit (бит целой части) — часть нормализованной мантиссы в формате с плавающей точкой. В этих форматах целый разряд расположен слева от двоичной точки. Значение этого разряда всегда 1, за исключением единственного случая: когда порядок минимален (смещенный порядок равен 0), то значение этого разряда есть 0. В расширенном формате разряд целой части явный; в обычном и двойном форматах — неявный, то есть в памяти реально не существует. См. также **Implicit Integer Bit.**

Internal Cache (внутренний кэш) — кэш-память на микросхеме процессора.

Interrupt (прерывание) — принудительная передача управления, вызванная сигналом от аппаратуры или выполнением команды INT n. Обработчики прерываний, вызываемые программно, работают как исключения.

Interrupt Descriptor Table (**IDT** — таблица дескрипторов прерываний) — массив дескрипторов шлюза для вызова обработчиков прерываний и исключений. Обработчики могут вызываться через шлюз задачи, шлюз прерывания или шлюз ловушки.

Interrupt Gate (шлюз прерывания) — дескриптор шлюза, используемый для вызова обработчика прерывания или исключения. Шлюз прерывания отличается от шлюза ловушки только влиянием на значение флага IF. Шлюз прерывания очищает данный флаг (блокирует прерывания) на время выполнения обработчика.

Invalid (недостоверная) — пустая. Например, недостоверные строки кэша не могут обеспечить кэш-попадания. Только достоверные строки кэша содержат данные и могут обеспечить кэш-попадание.

Invalid Operation (недопустимая операция) — условие исключения для модуля обработки операций с плавающей точкой, которое соответствует всем исключениям, не отвечающим другим условиям. Включает в себя: переполнение и отрицательное переполнение стека сопроцессора, NaN (не-число), неверная бесконечность, выход за допустимые границы, неподдерживаемый формат.

KNI (Katmai New Instructions) — дальнейшее развитие инструкций (команд) x86, в котором применяются потоковые инструкции **SIMD**, ускоряющие операции других видов, включая операции с плавающей запятой, первоначально не включенных в группу новых инструкций MMX. Katmai — это рабочее название Intel Pentium III.

Label (метка) — идентификатор места в тексте программы для ссылки на него. Места, именованные метками, включают точки входа процедур, начало блоков данных и базовые адреса таблиц дескрипторов.

Linear Address (линейный адрес) — 32-битовый адрес в большом сегментированном адресном пространстве (начиная с процессора i386). Если страничный обмен заблокирован, то линейный адрес транслируется в физический. Если же страничный обмен разрешен, то линейный адрес используется в качестве физического.

Linker — компоновщик. См. **Компоновка.**

Local Descriptor Table (**LDT** — локальная таблица дескрипторов) — массив дескрипторов сегмента для одной программы. Каждая программа может иметь свою собственную таблицу, может разделять ее с другой программой или не иметь вообще. В последнем случае программа использует глобальную таблицу дескрипторов — см. **GDT**.

Locked Instructions (команды блокирования) — команды считывания и записи в память, запрещающие другим устройствам становиться хозяевами системной шины между циклами считывания и записи. Такой механизм необходим для обеспечения надежной связи при мультипроцессорном режиме работы. Вызов механизма происходит при использовании командного префикса LOCK. Разрешается блокирование только команд, имеющих операнды назначения в памяти (для остальных команд использование префикса LOCK вызывает исключение недопустимой команды).

Logical Address (логический адрес) — число, используемое программой для ссылки на виртуальную память. Состоит из двух частей: селектора сегмента (16 разрядов) и смещения (16 или 32 разряда). Селектор сегмента используется для определения независимого защищенного адресного пространства (сегмента). Смещение указывает адрес внутри сегмента. При сегментации логические адреса преобразуются в линейные.

Long Integer (длинное целое) — формат целых в модуле обработки операций с плавающей точкой, состоящий из 64 битов памяти.

Long Real (длинное вещественное) — устаревшее название для 64-разрядного (двойного формата) вещественного числа.

Mantissa (мантисса) — термин, используемый для обозначения мантиссы числа с плавающей точкой.

Masked (маскированный) — термин, который можно применить к каждому из шести исключений в модуле обработки операций с плавающей точкой: IM, DM, ZM, OM, UM, PM. Исключение маскировано, если соответствующий бит слова управления установлен в 1. Если исключение замаскировано, то в случае генерации исключения, модуль обработки операций с плавающей точкой не будет генерировать прерывание, а произведет собственную обработку. См. **Unmasked**.

Memory Management (управление памятью) — процесс, состоящий из трансляции адреса и проверки защиты. Существует две формы управления памятью: сегментация и страничная организация памяти. Сегментация обеспечивает разбиение памяти на защищенные независимые адресные пространства (сегменты). Страничная организация памяти обеспечивает доступ к структурам данных, имеющим размер больше, чем размер доступного объема памяти, сохраняя их частично в оперативной памяти и частично на диске.

Microprocessor (микропроцессор) — см. **Processor**.

MMX (Multimedia Extensions) — расширение набора инструкций x86 от фирмы Intel (57 инструкций) с использованием технологии SIMD для ускорения вычислений, связанных с мультимедиа. См. также **3DNow!, KNI, SIMD**.

Mode (режим).

1. Одно из полей слова состояния сопроцессора, которое отвечает за «управление округлением» и «управление точностью». Для выполнения арифметических операций можно программно устанавливать, сохранять и восстанавливать значение этого поля.
2. См. **Real-Address Mode, Protected Mode, Virtual-8086 Mode, Supervisor Mode, User Mode**.

ModR/M Byte — байт, располагаемый за кодом команды и используемый для спецификации режима адресации операнда.

MPU (Micro Processor Unit — микропроцессорное устройство). См. **Processor**.

Multriprocessing (многопроцессорность, многопроцессорная обработка) — использование в системе более одного процессора для выполнения одного или нескольких процессов.

Multisegmented Model (многосегментная модель) — организация памяти, при которой различные сегменты преобразуются к различным диапазонам линейных адресов. Такая организация памяти использует сегментацию для защиты структур данных от нарушений, вызванных ошибками в программах.

Multitasking (многозадачность) — разделение времени процессора между несколькими программами, поочередное выполнение некоторого числа команд из каждой программы. При этом создается иллюзия одновременного выполнения этих задач.

NaN (Not a Number — не число) — некоторое значение, получаемое при операциях с плавающей точкой, которое не является ни числом, ни бесконечностью. NaN возвращается в качестве результата при обнаружении серьезных ошибок.

Near Pointer (ближняя ссылка) — ссылка на место в памяти без селектора сегмента; смещение. Используется для доступа к памяти, если селектор сегмента уже был загружен в процессор. Например, вызов одной процедуры из тела другой, при условии, что процедуры располагаются в одном сегменте кода.

Normal (нормальное) — представление числа в формате с плавающей точкой, в котором мантисса имеет один бит целой части (явный или неявный).

Normalize (нормализация) — преобразование денормализованного представления числа с плавающей точкой к нормальному представлению.

Offset (смещение) — 16- или 32-разрядное число, которое определяет место в памяти относительно базового адреса сегмента. Дескриптор сегмента кода программы определяет установлено ли по умолчанию 16- или 32-разрядное смещение. Префикс размера адреса позволяет отменить использование размера, установленного по умолчанию.

Operand (операнд) — данные в регистре или в оперативной памяти, которые считываются или записываются командой.

Operand-Size Prefix (префикс размера операнда) — префикс команды, который определяет размер операнда при 32-разрядном программировании. Операнды могут быть 8-ми, 16- или 32-разрядными. Размер операнда по умолчанию устанавливается битом D дескриптора сегмента кодов. При использовании префикса размера команды можно установить нужный размер операндов.

Overflow (переполнение). Для сопроцессора — это условие исключения для операции с плавающей точкой, в которой правильный ответ конечен, но его значение слишком велико, чтобы быть представленным в нужном формате. Эту разновидность переполнения (называемую числовым переполнением) нельзя смешивать с переполнением стека или с переполнением при целочисленных вычислениях, которое обычно не фиксируется и не обрабатывается.

Packed Decimal (упакованный десятичный) — целочисленный формат, поддерживаемый сопроцессором. Упакованное десятичное — это 10-байтовая величина, у которой 1 байт занимает знак, а остальные девять — 18 цифр в двоично-десятичном коде.

Page Directory (каталог страниц) — таблица страниц первого уровня. Процессор, начиная с i486, при страничной организации памяти использует двухуровневые таблицы страниц, где физический адрес, получаемый из таблицы первого уровня, является базовым адресом таблицы страниц второго уровня. Использование двухуровневой организации позволяет размещать таблицы второго уровня на диске.

Page Directory Base Register (PDBR — базовый регистр каталога страниц) — регистр процессора для хранения базового адреса каталога страниц; тоже самое что и регистр CR3. Так как содержимое регистра загружается из сегмента состояния задачи (TSS) во время переключения задач, а каждая задача может иметь свой собственный каталог страниц, то различные задачи могут иметь различные отображения виртуальных страниц в физические.

Page (страница) — блок последовательно расположенной памяти объемом 4 Кбайта; единица обмена, используемая при страничной организации памяти.

Page Table (таблица страниц) — таблица для преобразования линейных адресов в физические — см. **Page Directory**.

Page Table Entry (запись (элемент) таблицы страниц) — 32-битовая структура, используемая при страничной организации памяти. Включает физический адрес страницы и информацию о защите страницы. Значение элемента устанавливается программным обеспечением операционной системы и поддерживается аппаратурой страничной организации.

Paging (страничная организация памяти) — форма управления памятью для моделирования большого несегментированного адресного пространства с использованием части дисковой памяти и фрагментированного адресного пространства. Обеспечивает доступ к структурам данных, имеющим размер больше, чем размер доступного объема оперативной памяти, сохраняя их частично в оперативной памяти и частично на диске.

Physical Address (физический адрес) — адрес, передаваемый на локальную шину процессора.

Physical Memory (физическая память) — диапазон адресов, передаваемый на локальную шину процессора; аппаратная реализация памяти.

Precision (точность) — количество значимых битов мантиссы в представлении числа с плавающей точкой.

Precision Control (управление точностью) — опция, запрограммированная через слово управления сопроцессора, которая разрешает выполнять арифметические операции для данных с плавающей точкой с уменьшенной точностью. Так как никаких преимуществ по скорости получения результата эта опция не дает, то она используется только для совместимости со стандартом IEEE-754 и другими вычислительными системами.

Precision Exception (исключение нарушения точности) — исключение модуля обработки операций с плавающей точкой, которое происходит, если при вычислениях получен неточный результат. Эта исключение обычно маскируется и игнорируется. Оно используется обычно только в случае, когда пользователю необходимо знать точность результата. По стандарту IEEE-754 называется неточностью.

Previlige Level (уровень привилегированности) — параметр защиты, применяемый к сегментам и селекторам сегментов. Существует четыре уровня привилегированности от 0 (наиболее привилегированный) до 3 (наименее привилегированный). Привилегированность 0-уровня используется для системного программного обеспечения, такого как операционная система, системные драйверы. Привилегированность 3-уровня используется для прикладных программ. Некоторые системные программы, например, драйверы устройств, могут иметь промежуточный уровень привилегированности 1 или 2.

Processor (процессор) — часть вычислительной системы, исполняющая команды; также называется микропроцессор, ЦПУ или МПУ.

Protected Mode (защищенный режим) — режим работы, при котором доступна вся 32-битовая архитектура процессора.

Protection (защита) — механизм, используемый для защиты операционной системы и прикладных программ от ошибок в прикладных программах. Может использоваться для определения адресного пространства, доступного программе, типа доступа к определенному месту памяти и уровня привилегированности. Любое нарушение генерирует исключение общей защиты. Применяется к страницам или сегментам.

Pseudo-Descriptor (псевдо-дескриптор) — 48-разрядный операнд памяти, используемый для загрузки или сохранения базового регистра таблицы дескрипторов.

Pseudozero (псевдонуль) — одна из множества специальных величин расширенного вещественного формата. Множество состоит из чисел с нулевой мантиссой и с порядком, который не может состоять только из нулей или только из единиц. Псевдонули не создаются сопроцессором, но обрабатываются правильно как операнды.

Quadrword (учетверенное слово) — 64-битовый операнд. Команда CDQ используется для преобразования двойного слова в учетверенное слово. Учетверенное слово, хранящееся в регистрах EDX и EAX, может быть делимым, при использовании делителя в формате двойного слова.

Quiet NaN (QNAN — безответное (тихое) **NaN)** — значение NaN в формате с плавающей точкой, у которого большая часть битов мантиссы дробной части равна 1. Для удобства, эти QNaN могут использоваться в некоторых случаях без генерации исключений.

RAM (Random Access Memory — память с произвольным доступом) — энергозависимая память с произвольным доступом для чтения и записи, которая используется, главным образом, для хранения программ и данных во время их выполнения на ПК. Отличается от **ROM** и флэш-памяти.

Re-entrant (повторный вызов, реентерабельность) — разрешение программе повторно вызвать саму себя (рекурсия). Для некоторых задач, таких как операции над иерархическими структурами данных, рекурсивные процедуры являются простым и эффективным решением. Процессор i486 (и более поздние) поддерживает рекурсивные процедуры, но не рекурсивные задачи. Задача не может быть рекурсивной, так как для хранения состояния процессора она имеет только один сегмент состояния задачи (**TSS**). Процедуры сохраняют состояние процессора в стеке, поэтому могут иметь бесконечную рекурсию (теоретически).

Real-Address Mode (режим реальной адресации) — режим работы, который обеспечивает эмуляцию архитектуры процессора 8086; также называется реальным режимом. В этом режиме процессоры, начиная с i386, работают как быстрый процессор 8086. Архитектурные дополнения по защите и многозадачности не доступны в этом режиме. При инициализации, следующей после команды RESET или после включения питания, процессор начинает работать в режиме реальной адресации.

Real (вещественное) — любая конечная величина (отрицательная, положительная, нуль), которая может быть представлена десятичным расширением (возможно бесконечным). Термин также применяется к числу с плавающей точкой, которое представляет собой вещественную величину.

Requested Privilege Level (**RPL** — запрашиваемый уровень привилегированности) — уровень привилегированности для селектора сегмента в защищенном режиме. Если RPL меньше текущего уровня привилегированности (**CPL**), то доступ к сегменту памяти с более высоким приоритетом будет закрыт с генерацией особой ситуации ¦ошибка общей защиты¦. Это предохраняет привилегированное программное обеспечения от вмешательства прикладных программ. Например, прикладной программе, загружающейся с диска, не разрешается переписывать операционную систему.

Reset (перезагрузка) — см. **Initialization**.

ROM (Read-Only Memory —постоянное запоминающее устройство, предназначенное только для чтения). Этот термин применяется к нескольким видам памяти: электронной, CD, DVD. При применении к электронной памяти он означает чипы памяти, хранящие программы и данные, которые не изменяются и должны всегда быть доступны для процессора.

Segment (сегмент) — независимое защищенное адресное пространство. Программа может иметь не более 16383 сегментов, каждый из которых имеет объем до 4 Гбайтов.

Segment Descriptor (дескриптор сегмента) — 64-битовая структура данных, используемая при сегментации памяти. Включает базовый адрес сегмента, его размер (границы),. тип и информацию по защите. Начальное значение устанавливается операционной системой, доступ поддерживается аппаратными средствами по управлению сегментацией.

Segment-Override Prefix (префикс замены (подавления) сегмента) — префикс команды, который заменяет используемый по умолчанию сегмент. Существует шесть префиксов переопределения сегмента, по одному для каждого из сегментов CS, SS, DS, ES, FS и GS.

Segment Selector (селектор сегмента) — 16-разрядное число, используемое для спецификации области памяти (сегмента). Биты 3-15 являются индексом в таблице дескрипторов (**Descriptor Table**). Бит 2 определяет, какая из таблиц дескрипторов используется — локальная (**LDT**) или глобальная (**GDT**). Биты 0 и 1 указывают запрашиваемый уровень привилегированности (**RPL**), который может понижать привилегированность доступа для дополнительной проверки защиты.

Segmentation (сегментация) — форма управления памятью, разбивающая память на независимые, защищенные адресные пространства. Сегментация помогает отладке программ, сигнализируя об ошибках немедленно, до того как проявится ее действие. Сегментация делает программы более надежными, ограничивая нарушения, которые могут быть вызваны необнаруженными ошибками. Она увеличивает адресное пространство, доступное программе, предоставляя 16383 сегментов, каждый из которых может иметь объем до 4 Гбайт (начиная с процессоров i386).

Set-Associative (множественно-ассоциативный) — форма организации кэша, при которой блок данных «привязан» к месту в основной памяти, но не имеет полностью определенного места в кэше. Множественно-ассоциативная организация является компромиссом между прямо преобразуемой организацией (данные по данному месту в памяти, имеют только одно возможное место в кэше) и полностью ассоциативной организаций (данные из любого места в памяти могут быть размещены в любом месте кэша). «Множественно-ассоциативный n-связный» кэш разрешает данным по любому месту в основной памяти быть размещенными в одном из n мест в кэше. Оба и буфер ассоциативной трансляции (**TLB**), и внутренний кэш процессора, начиная с i486, имеют четырехсвязную множественно-ассоциативную организацию.

Short Integer (короткое целое) — формат для целых чисел, поддерживаемый сопроцессором. Имеет размер 32 бита. Короткое целое — не самый короткий формат целых чисел, допустимый для модуля обработки операций с плавающей точкой (самый короткий формат занимает 16 разрядов).

Short Real (короткое вещественное) — устаревший термин обычного 32-разрядного формата вещественного числа в сопроцессоре.

SIB Byte — байт, следующий за кодом операции и байтом **modR/M** (или вместо байта **modR/M**) в процессорах, начиная с i386. Он используется для расширения способа адресации при 32-разрядном программировании.

Sign Extension (распространение знака) — преобразование данных к большему формату, в котором пустые разряды заполняются значением знакового бита. Такая форма преобразования сохраняет значения знаковых целых, расширяя их формат. См. **Zero Extension**.

Signaling NaN (**SNaN** — сигнальный **NaN**) — значение NaN с плавающей точкой, которое вызывает исключение недопустимой операции всякий раз, когда оно встречается в вычислениях или сравнениях, даже при сравнениях несравнимых величин.

Significand (мантисса) — часть числа с плавающей точкой, которая состоит из старших значащих ненулевых разрядов числа, если число было записано в неограниченном двоичном формате. В мантиссу входят бит целой части и дробь (нормализованное представление мантиссы). В обычном и двойном формате бит целой части является неявным (невидимым). Считается, что мантисса имеет двоичную точку после целого бита. Двоичная точка передвигается в соответствии со значением порядка.

SIMD (Single Instruction, Multiple Data — один поток команд и множество потоков данных) — технология, используемая в **MMX** и **KNI**. См. также **3DNow!**

Single Extended (обычный расширенный) — формат для чисел с плавающей точкой, принятый стандартом **IEEE-754**, который дает большую точность, чем обычный формат. Сопроцессор допускает команды как для обычного расширенного формата, так и для двойного расширенного формата.

Single Format (обычный формат) — формат для чисел с плавающей точкой, поддерживаемый сопроцессором. Состоит из знака, 8-битового смещенного порядка, неявного бита целой части и 23-битовой мантиссы — всего 32 значащих бита.

Stack Fault (сбой стека) — особый случай исключения недопустимой операции, который указывается битом SF слова состояния. Обычно имеет место при переполнении или отрицательном переполнении стека сопроцессора.

Stack Frame (кадр стека) — стековое пространство, используемое в процедуре. Включает параметры, адреса возврата, значения регистров, временную память и данные, используемые процедурой.

Stack Segment (стековый сегмент) — сегмент данных, используемый для хранения стека. Может быть расширяемым вниз, то есть расти по направлению к младшим адресам. Тип информации, хранимой в сегменте, определятся дескриптором сегмента.

Status Word (**SW** — слово состояния) — 16-разрядный регистр, значение которого может быть установлено вручную, но на который влияют сторонние эффекты команд сопроцессора. Содержит коды условий, ссылку на вершину стека сопроцессора, бит занятости, а также флаги исключений.

String (строка) — последовательность байтов, слов или двойных слов, которая может начинаться с любого байта памяти.

Supervisor Mode (режим супервизора) — уровень применительно к страницам оперативной памяти. Страничная организация памяти распознает только два уровня привилегированности: режим супервизора и пользовательский режим. Программа из сегмента с уровнем привилегированности 0, 1, 2 выполняется в режиме супервизора.

Table (таблица) — массив данных, состоящий из элементов одинакового размера.

Tag Word (слово тега) — 16-разрядный регистр сопроцессора, который автоматически устанавливается модулем обработки операций с плавающей точкой. Этот регистр содержит информацию о том, чем занят (или не занят вообще) каждый регистр стека сопроцессора.

Tag (тег) — часть строки кэша, которая содержит адресную информацию, используемую для определения наличия или отсутствия операнда памяти в данной строке кэша.

Task Register (регистр задачи) — регистр для хранения селектора сегмента текущей задачи. Селектор содержит указатель на сегмент состояния задачи (TSS). Подобно сегментным регистрам регистр задачи имеет видимую и невидимую части. Видимая часть хранит селектор сегмента, невидимая часть содержит информацию из дескриптора сегмента для TSS.

Task Segment State (TSS — сегмент состояния задачи) — сегмент, используемый для хранения состояния процессора во время переключения задач. Если используется разделяемое адресное пространство ввода-вывода (I/O) , то TSS сохраняет значение битов разрешения, которые управляют доступом к пространству I/O. Операционная система может определить дополнительные структуры, существующие в TSS.

Task Switch (переключение задач) — передача управления между задачами; контекстный переключатель. В отличие от вызовов процедур, которые сохраняют только содержимое основных регистров, переключатель задач сохраняет большую часть состояния процессора. Например, перезагружаются регистры, используемые для трансляции адресов, поэтому каждая задача может иметь различные преобразования адресов от логических к физическим.

Task (задача) — выполняемая или отложенная программа в многозадачной системе.

Temporary Real (временное вещественное) — устаревший термин для 80-разрядного расширенного формата.

Tiny (очень маленький, «крошечный»).

- Число с плавающей точкой, настолько близкое к нулю, что его порядок меньше наименьшего порядка, который может быть получен в соответствующем формате.
- Модель памяти, используемая для получения com-файла в 16-разрядном программировании.

Top (вершина) — 3-битовое поле слова состояния (SW, SWR), которое указывает, какой из регистров сопроцессора является текущей вершиной стека.

Transcendental (трансцендентный) — один из классов функций, которым соответствуют полиномиальные формулы. Модуль обработки операций с плавающей точкой поддерживает тригонометрические, степенные и логарифмические функции.

Translation Lookside Buffer (TLB — буфер ассоциативной трансляции) — кэш-память для элементов таблиц страниц. В обычных системах около 90% ссылок на элементы таблицы могут быть получены по информации из буфера.

Trap (ловушка) — исключение, сообщение о котором возникает на границе команд сразу же после команды, вызвавшей это исключение.

Trap Gate (шлюз ловушки) — дескриптор шлюза, используемый для вызова обработчика исключений. Шлюз ловушки отличается от шлюза прерывания только тем, что последний воздействует на флаг IF. В отличие от шлюза прерывания, который очищает этот флаг (блокирует прерывания) на время выполнения обработчика прерывания, шлюз ловушки оставляет данный флаг без изменения.

Two's Complement (двоичное дополнение) — метод представления целых. Если самый старший бит есть 0, то число считается положительным, а оставшиеся биты представляют значение числа. Если же самый старший бит есть 1, то число — отрицательное и оно представляется в формате дополнительного кода. Например, число +4 имеет двоичный эквивалент 0000 0100 (размер 8 бит), а -4 — 11111100. Если их сложить, то и в десятичном, и в двоичном формате получается 0.

Unbiased Exponent (несмещенный порядок) — истинная величина, которая сообщает насколько далеко и в каком направлении передвигать двоичную точку мантиссы. Например, если в обычном формате порядок равен 131, то для получения несмещенного порядка нужно вычесть смещение, равное 127. Искомое значение есть +4. То есть вещественное число есть мантисса, сдвинутая вправо на 4 бита.

Unmasked (немаскированное) — термин, который можно применять к каждому из шести исключений сопроцессора: IM, DM, ZM, OM, UM, PM. Исключение не маскировано, если в соответствующий бит управляющего слова (**CW, CWR**) установлен нуль. В этом случае сопроцессор при возникновении исключения будет генерировать прерывание. Вы можете разработать собственную программу обработки исключений. См. **Masked**.

Unsupported Format (неподдерживаемый формат) — любой формат, не распознаваемый модулем обработки операций с плавающей точкой. Некоторые форматы поддерживаются только сопроцессорами 8087 и i287, а именно: псевдо-NaN, псевдобесконечный и ненормальный.

USE16 — директива ассемблера для назначения 16-разрядных сегментов кода и данных.

USE32 — директива ассемблера для назначения 32-разрядных сегментов кода и данных.

User Mode (пользовательский режим) — уровень привилегированности, применительно к страницам прикладных программ. При страничной организации памяти существует только два уровня привилегированности: режим супервизора и режим пользователя. Программа из сегмента с привилегированностью 3 выполняется в пользовательском режиме.

Valid (достоверный). В достоверные строки кэша загружаются данные, обеспечивающие кэш-попадания. Недостоверные строки кэша не могут обеспечить кэш-попадания.

Virtual Memory (виртуальная память) — модель памяти для прикладных программ; организация памяти, поддерживаемая аппаратными средствами по управлению памяти и операционной системой. Для процессоров, начиная с i386, виртуальная память реализуется через сегментацию и страничную организацию памяти. Сегментация — механизм, предоставляющий независимое защищенное адресное пространство — см. **Segmentation**. Страничная организация памяти обеспечивает доступ к структурам данных, имеющим размер больше, чем размер доступного объема памяти, сохраняя их частично в оперативной памяти и частично на диске — см. **Paging**.

Virtual-8086 Mode — режим выполнения, эмулирующий архитектуру процессора 8086. В отличие от режима абсолютной адресации совместим с многозадачностью.

VxD (Virtual Device Driver — виртуальный драйвер х-устройства) — небольшие программы, которые наращивают операционную систему на самом низком ее уровне, и, следовательно, могут делать все, что только возможно. Они имеют прямой доступ к любой части аппаратуры. Например, VKD может являться виртуальным драйвером клавиатуры, а VDD — виртуальным драйвером дисплея.

WDM (Windows Driver Model — модель драйвера Windows) — последняя (на момент написания книги) спецификация Microsoft по созданию драйверов устройств. Драйверы устройств, написанные в соответствии с этим стандартом, будут работать как в Windows-98, так и в Windows-2000.

Windows SE — версия Microsoft Windows, предназначенная для использования в карманных ПК.

Windows — общее название для наиболее популярного семейства операционных систем ПК компании Microsoft. Это семейство включает в себя первоначальные программы Windows 1.0-3.11, которые являлись графическими оболочками интерфейса пользователя и загружались из DOS. Windows-95 и Windows-98 были уже самостоятельными операционными системами с графическим интерфейсом.Windows NT и Windows-2000 — полностью 32-разрядные операционные системы, рассчитанные на профессиональное использование в составе рабочих станций и серверов. Windows Me продолжила линейку Windows 9x и тоже стала полностью 32-разрядной операционной системой.

Wintel — так называют любой компьютер, основанный на архитектуре процессора Intel x86 и некоторой версии Windows.

Word (слово) — 16-битовый размер памяти. Процессор x86 позволяет словам начинаться с любого байта, однако, следует быть осторожными, так как слово может пересечь границу между двумя двойными словами в физической памяти.

Word Integer (целое слово) — формат для целых, поддерживаемый процессором x86. Их двоичное представление занимает 16 разрядов.

x86 — название семейства процессоров Intel, используемых в различных поколениях ПК (начиная с 8086, 8088, 80186, 80286, 386, 486 и кончая именованными процессорами — Pentium). Это имя включает в себя также и все совместимые процессоры, производимые AMD, Cyrix и другими компаниями.

Zero Divide (деление на нуль) — условие вызова исключения, при котором входное число с плавающей точкой конечно, но правильный ответ, даже с неограниченным порядком, имеет бесконечное значение.

Zero Extension (нулевое расширение, распространение нуля) — преобразование данных к большему формату, при котором пустые битовые позиции заполняются нулями. Это форма для преобразования беззнаковых целых. См. **Sign Extension**.

Адрес порта — ячейка памяти, которую использует процессор для манипуляции с данными через физический порт ввода-вывода.

Асинхронный — два или более события, не синхронизированных по времени. В контексте передачи данных, это процесс, при котором данные получаются или отправляются на разной скорости.

АЛУ (арифметико-логическое устройство) — см. **ALU.**

Время доступа — см. **Время ожидания**.

Время ожидания — задержка между моментом запуска команды и моментом, когда она начнет действовать.

Действительное число — любое число (положительное, отрицательное или нуль), включающее как целую часть, так и дробную.

Драйвер устройства — программа, которая позволяет компоненту аппаратного обеспечения или самой операционной системе активизировать и использовать устройство. См. также **VxD**.

Защищенный режим — режим, позволяющий процессорам, начиная с i386, выполнять 32-разрядные команды и использовать плоскую (Flat) модель памяти. Отличается от **реального режима**. См. также **Protected Mode.**

Компиляция — процесс преобразования инструкций компьютерного языка (например, Ассемблера, C/C++) в машинный код и сохранения этого кода в объектном файле. На этом этапе могут быть отловлены синтаксические ошибки программирования и сделан файл листинга.

Компоновка (сборка) — процесс построения исполняемого файла из объектных модулей. См. **Linker**.

Кэш — временное место хранения данных. См. **Cache.**

Объектно-ориентированное программирование (ООП) — стиль программирования, базирующийся на методах, классах, объектах, наследовании, инкапсуляции и других элементах, позволяющий существенно повысить скорость разработки программ путем использования шаблонов, наследования и переопределения методов.

Операционная система — программное обеспечение, которое управляет и распределяет ресурсы компьютера, облегчая пользователю и программисту процесс взаимодействия с компьютером.

Основная память — некоторый объем **RAM** или **ROM**, к которому может напрямую обращаться процессор. Такая память находится на материнской плате, но может находиться и на видеокарте или другой карте расширения.

Память — пространство, составляемое чипами **RAM** различных типов, в котором компьютер выполняет свою работу.

Порт ввода-вывода — аппаратный интерфейс (см. **Interface**) между процессором и некоторым внешним устройством. Каждый порт ввода-вывода имеет **адрес порта** или диапазон смежных адресов (нижний из которых называется базовым адресом). Устройства, подключаемые к портам ввода-вывода, могут находиться как внутри ПК, так и быть к нему подключенными извне.

Производительность — скорость обработки данных компьютером. Обычно выражается в бит/с.

Реальный режим — режим работы, в котором современные Intel-совместимые процессоры ведут себя подобно процессору 8086/8088 и способны адресовать только 1М байт оперативной памяти. См. **Real-Address Mode.**

Синхронный — два или более событий, происходящих одновременно. При передаче данных они передаются или принимаются с фиксированной скоростью.

Центральный процессор (ЦП) — устройство внутри ПК, которое выполняет основную работу по обработке информации. См. также **CPU, ALU, MPU, Microprocessor, Processor, АЛУ, Coprocessor, FPU.**

Предметный указатель

Символы

А

Б

В

Д

З

И

К

П

Р

С

Иностранные термины

O

P

R

S

T

V

W

Научное издание

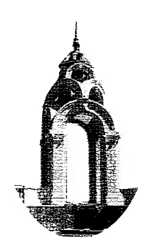
ГОЛУБЬ Н.Г.
Искусство программирования на Ассемблере.
Лекции и упражнения
2-е изд., исправленное и дополненное

Заведующий редакцией *С.Н.Козлов*
Верстка *А.С. Ребенок*
Главный дизайнер *О.А.Шадрин*

ООО «ДиаСофтЮП», 196105, Санкт-Петербург, пр. Ю.Гагарина, д. 1, ком. 108.
Лицензия №000328 от 9 декабря 1999 г.

Сдано в набор 30.09.2002. Подписано в печать 25.10.2002. Формат 70×100/16.
Бумага типографская. Гарнитура Таймс. Печать офсетная. Печ.л. 41.
Тираж 3000 экз. Заказ № 893

Отпечатано с готовых диапозитивов
в ФГУП ордена Трудового Красного Знамени «Техническая книга»
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций
198005, Санкт-Петербург, Измайловский пр., 29.



Подходите творчески к процессу программирования

- ***Делай с нами***: постепенное, от простого к сложному, изучение теоретического материала, примеров и законченных программ (часть 1, 16 глав, более 50 программ с различными вариантами их реализации)
- ***Делай, как мы***: изучение на компьютере реализаций типовых вариантов лабораторных работ, нахождение и устранение ошибок (часть 2, 10 лабораторных работ)
- ***Делай лучше нас***: выполнение заинтересовавших вас вариантов лабораторных работ (чем больше, тем лучше – часть 2, 10 лабораторных работ по 60 вариантов в каждой)

Дискета, поставляемая отдельно, содержит исходные коды всех программ, рассмотренных в книге

На сайте издательства "ДиаСофт" по адресу **www.diasoft.kiev.ua** находятся все необходимые исходные тексты программ, которые рассмотрены в первой и второй частях книги, а также набор инструментальных и вспомогательных средств (всего около 4 Мб).

Об авторе

Голубь Надежда Григорьевна — кандидат технических наук, доцент кафедры программного обеспечения *Национального аэрокосмического университета* (ХАИ), г. Харьков.

Область профессиональных интересов: языки программирования (знает более 20 основных алгоритмических языков и их диалектов, в том числе такие известные, как ***Algol***, ***PL/1***, ***Fortran***, ***Pascal***, ***C/C++***, ***Ada***, ***Java***, ***Smalltalk***, ***Prolog***, ***LISP***), системное программирование и компиляторы, *Internet*-программирование, история развития аппаратного и программного обеспечения компьютеров.

В настоящее время осваивает дистанционное обучение — имеет около 10 тысяч виртуальных учеников со всего мира и свой собственный сайт **http://www.anriintern.com/computer/c++/** — "Язык C++ для начинающих", который входит в состав портала бесплатного дистанционного образования "Anri Education Systems" (**http://www.anriintern.com/**).

Связаться с автором можно по адресу: **n_golub@ukr.net**



Книга окажется полезной для преподавателей и студентов, профессионально изучающих программирование



Украина, Киев, 2003

ISBN 5-93772-056-3

